Verbindung mehrerer uMundo Workspaces

Connecting uMundo Workspaces
Bachelor-Thesis von Thilo Molitor
Tag der Einreichung:

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Tim Grube

TECHNISCHE UNIVERSITÄT DARMSTADT

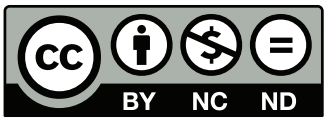
Fachbereich Informatik
Telecooperation Group

Verbindung mehrerer uMundo Workspaces
Connecting uMundo Workspaces

Vorgelegte Bachelor-Thesis von Thilo Molitor

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Tim Grube

Tag der Einreichung:



Erklärung zur Bachelor-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 1. Januar 2015

(Thilo Molitor)

Abstract

uMundo ist ein leichtgewichtiges Publish/Subscribe Framework mit einer einfach zu nutzenden API. Die automatische Discovery der einzelnen Publish/Subscribe Teilnehmer funktioniert allerdings nur innerhalb eines Subnetzes (uMundo Workspace genannt), da Multicast verwendet wird. Soll die Kommunikation über Routergrenzen hinweg funktionieren, so ist dies nicht ohne weiteres möglich. Die Umsetzung einer Verbindung mehrerer uMundo Workspaces über Routergrenzen hinweg ist daher die Aufgabe, der sich diese Bachelorarbeit widmet.

Die Bereitstellung von Sensordaten, Videostreaming, die Steuerung eines Smart-TVs oder der Heizung; all das lässt sich plattformunabhängig und in vielen verschiedenen Programmiersprachen mit uMundo realisieren. Was ist aber nun, wenn ich meinen Smart-TV von meinem Arbeitsplatz aus steuern möchte um beispielsweise eine interessante Sendung aufzunehmen? Was, wenn ich mir den Videostream der Überwachungskamera vom Urlaub aus ansehen möchte oder die Rollläden von dort aus steuern will? All diese Szenarien sind mit dem aktuellen Framework noch nicht umsetzbar. Es fehlt eine Brücke, eine Verbindung zwischen verschiedenen Netzwerken.

Eine solche Brücke lässt sich über ein VPN realisieren. Dieses Konzept birgt allerdings einige Nachteile. Verwendet man einen dedizierten application-level Proxy (in dieser Arbeit „Bridge“ genannt), so lassen sich diese Nachteile umgehen. Die Entwicklung einer solchen Bridge ist daher Teil dieser Bachelorarbeit. Möchte man zusätzlich auch noch den (fast einzigen) Vorteil einer Brücke via VPN erhalten (die Möglichkeit zur Authentifizierung und Verschlüsselung der Daten), so kann die Bridge zusätzlich durch einen VPN-Tunnel hindurch aufgebaut werden.

Inhaltsverzeichnis

1 Motivation

Intelligente Geräte wie Smartphones oder SmartTVs gehören mittlerweile zum Alltag unserer Gesellschaft.

Damit diese smarten Geräte auch ihr volles Potential ausschöpfen können, müssen sie in der Lage sein, miteinander zu kommunizieren. Der smarte Wecker muss mit der smarten Kaffeemaschine „reden“ können und ihr „sagen“, dass der Besitzer gerade aufgestanden ist und in ein paar Minuten seinen morgendlichen Kaffee braucht. Der SmartTV muss mit dem Smartphone „reden“ können und so den Benutzer daran erinnern, dass seine Lieblingssendung demnächst beginnen wird.

Im Internet und in der Literatur finden sich einige Frameworks, die versprechen, genau diese Art der Kommunikation über eine einfache API zugänglich zu machen. Eines dieser Frameworks ist uMundo, welches als Grundlage für die hier vorliegende Bachelorarbeit dient.

uMundo ist ein Publish/Subscribe Framework, das ein einfaches Anbieten von Daten auf sogenannten *Channels* erlaubt. Ein oder mehrere Publisher binden sich dafür an einen *Channel* und können fortan beliebige Daten auf diesem *Channel* publizieren.
Um diese Daten dann zu empfangen, können sich die Subscriber ebenfalls an diesen *Channel* binden und erhalten dann die Daten aller Publisher, die auf diesem *Channel* publizieren.

Trotz all seiner Funktionen unterliegt uMundo allerdings einer Einschränkung: Es werden, aufbauend auf dem Domain Name System (DNS), Multicast DNS (mDNS) [CK13b] und wiederum darauf aufbauend DNS Service Discovery (DNS-SD) [CK13a] für die Discovery von Publishern und Subscribern genutzt. Leider funktioniert mDNS nur im lokalen Subnetz (in uMundo Terminologie auch *(Smart) Workspace* genannt).

Sollen mehrere uMundo Geräte über Routergrenzen hinweg verbunden werden, beispielsweise zwei Haushalte über das Internet, so ist dies nicht ohne Weiteres möglich.

Was ist aber nun, wenn ich meinen Smart-TV von meinem Arbeitsplatz aus steuern möchte, um beispielsweise eine interessante Sendung aufzunehmen? Was, wenn ich mir den Videostream der Überwachungskamera vom Urlaub aus ansehen möchte oder die Rollläden von dort aus steuern will? All diese Szenarien sind mit dem aktuellen Framework noch nicht umsetzbar. Es fehlt eine Verbindung zwischen verschiedenen Netzwerken, die uMundo von einem lokalen Tool zu einem global einsetzbaren Framework machen würde.

Die Umsetzung einer solchen Verbindung ist nun die Aufgabe, der sich die hier vorliegende Bachelorarbeit widmet. Sie gliedert sich wie folgt: In Kapitel 2 werden die Grundlagen von Publish/Subscribe und uMundo erklärt, in Kapitel 3 werden Alternativen zu uMundo und Publish/Subscribe aufgezeigt, in Kapitel 4 werden dann die Lösungskonzepte beschrieben, Kapitel 5 beschreibt die Implementierung dieser Konzepte und Kapitel 6 fasst den Inhalt dieser Arbeit zusammen.

2 Grundlagen

In der vorliegenden Bachelorarbeit werden einige Systeme und Software Engineering Paradigmen häufig erwähnt. Diese Systeme und Paradigmen sollen im Folgenden näher erläutert werden.

2.1 Publish/Subscribe

Publish/Subscribe ist eine Kommunikationsunterart des Message Queue Paradigmas. Bei diesem Paradigma werden Informationen zwischen Prozessen oder auch zwischen Threads über sogenannte Message Queues ausgetauscht. Die Queues sorgen hierbei dafür, dass die Kommunikation zwischen den Teilnehmern asynchron läuft. Sender und Empfänger müssen nicht gleichzeitig auf die Queue zugreifen, um Informationen austauschen zu können.
Eine Queue ist für gewöhnlich unidirektional. Für eine bidirektionale Kommunikation werden also normalerweise 2 Queues benötigt. Außerdem sorgen die Queues dafür, dass alle Nachrichten trotz der asynchronen Kommunikation noch in der Reihenfolge beim Empfänger ankommen, in der sie vom Sender abgeschickt wurden.

Publish/Subscribe sorgt dafür, dass nur noch eine lose Kopplung zwischen dem Sender von Daten (Publisher) und dem Empfänger (Subscriber) existiert. Dafür wird das Management der einzelnen Queues und die Auswahl der zu übermittelnden Daten aus den einzelnen Queues durch ein entsprechendes Framework unterstützt, welches oft auch die Kommunikation über ein Netzwerk ermöglicht.
Wie beim Message Queueing auch, werden bei Publish/Subscribe die Daten paketweise als atomische Messages übertragen. Soll ein Empfänger nicht alle Daten bekommen, die via Publish/Subscribe verteilt werden, so müssen die Daten bzw. Messages gefiltert werden. Hierfür existieren verschiedene Ansätze:

- **Topic-based systems**
  Hier werden die einzelnen Messages mit einem Topic markiert. Der Subscriber kann dann eingehende Messages anhand dieser Markierung filtern. So muss er nur noch Messages bearbeiten, für deren Topic er sich interessiert.

- **Content-based systems**
  Bei diesem Ansatz werden die einzelnen Messages nach ihrem Inhalt gefiltert und nur die Messages an den Subscriber übermittelt, die den festgelegten Inhaltseinschränkungen genügen.

- **Hybrid systems**
  Bei diesen Systemen werden beide oben genannte Filterungen unterstützt. Ein Subscriber kann also die gewünschten Nachrichten sowohl über die Angabe des Topics als auch über das Setzen eines Filterkriteriums für den Inhalt genauer spezifizieren.

Oftmals nutzen Publish/Subscribe Systeme für das Übermitteln der einzelnen Messages einen Message Broker, also einen Vermittler, der die von den Publishern empfangen Messages zwischenspeichert und dann an die Subscriber weitergibt. Ein solcher Message Broker übernimmt dabei oft auch die Filterung der Messages selbst, bevor er diese an die Subscriber weitergibt.
Zwingend notwendig ist ein solcher Message Broker allerdings nicht. uMundo baut beispielsweise auf direkter Kommunikation zwischen den beteiligten Kommunikationspartnern auf und nutzt keinen zentralen Message Broker.

2.2 RTP Streaming

Das Real-Time Transport Protocol (RTP) [SCFJ03, SC03, PW10, Ter13] ist ideal, um Daten in Paketen zu übertragen, deren Paketinhalt nach kurzer Zeit schon uninteressant geworden ist und durch neuere Daten ersetzt werden soll. Kommt ein solches Paket nicht oder verzögert beim Empfänger an, so soll dieses einzelne Paket nicht die Zustellung der restlichen Pakete verzögern.
Daher baut RTP auf dem User Datagram Protocol (UDP) [Pos80] auf, welches im Gegensatz zum Transmission Control Protocol (TCP) [Pos81] [JBB92] paketorientiert ist.

Sowohl Sensordaten als auch Sprache oder Video lassen sich dadurch über RTP viel besser verteilen als über TCP, auf welchem die meisten Publish/Subscribe Systeme aufbauen.

2.3 uMundo

uMundo ist (wie bereits erwähnt) ein Publish/Subscribe System, das unter der 3-Clause BSD Lizenz [3cl99] steht. Es ist *topic-based*, wobei die Topics in uMundo Terminologie *Channels* heißen. Die Kommunikation zwischen Publishern und Subscribern erfolgt direkt, ein Message Broker ist bei uMundo nicht zwischen geschaltet. uMundo unterstützt die Plattformen Windows, Linux, Mac OSX, iOS und Android (sowohl 32bit als auch 64bit)[1] und bietet vorkompilierte Software Developement Kits (SDKs) für Mac OS X, Linux und Windows.

Die einzelnen Publisher und Subscriber können innerhalb eines Prozesses zu sogenannten *Nodes* zusammengefasst werden, die sich im lokalen Subnetz via Avahi [ava] oder Bonjour [bon] finden können. Beide Discovery-Mechanismen bauen auf DNS-SD und mDNS auf.

uMundo unterstützt neben der „normalen“ Publish/Subscribe Kommunikation, die über TCP abgewickelt wird - die Daten gehen daher nicht verloren und kommen immer in der richtigen Reihenfolge an - auch eine Kommunikation über RTP.
Hier können/dürfen Daten verloren gehen oder in der falschen Reihenfolge ankommen. Das Timestamp-Feld und das Sequenznummer-Feld in den Metadaten der empfangenen uMundo Message zeigen dabei an, ob das Paket eventuell in der falschen Reihenfolge angekommen ist.

[1] Siehe uMundo Github Repository unter `https://github.com/tklab-tud/umundo`

uMundo ist in drei Hauptkomponenten unterteilt:

- **umundo.core**
  Diese Komponente bildet die Basis von uMundo und bietet das Publish/Subscribe Pattern in Kombination mit einer Discovery über eine einfache API an. Hierbei wird zusätzlich zur „normalen“ Publish/Subscribe Kommunikation auch eine spezielle für Streaming optimierte Publish/Subscribe Kommunikation angeboten, die über das Real-Time Transport Protocol (RTP) [SCFJ03] realisiert wird. Diese Komponente unterstützt die Programmiersprachen C++, C#, Java, Python, Perl, PHP und Objective-C.

- **umundo.s11n**
  Diese Komponente bietet eine Serialisierung auf Basis von Googles *Protocol Buffers*[2] [pro] an.
  Das bedeutet, dass beliebige Objekte und andere Datenstrukturen der unterstützten Programmiersprachen in eine sequenzielle Form gebracht werden können. Diese Daten können dann als Byte-Array über uMundo verschickt und auf der Gegenseite wieder zur ursprünglichen Struktur zusammengesetzt werden. Diese Komponente unterstützt die Programmiersprachen C++, C#, Java und Objective-C.

- **umundo.rpc**
  Aufbauend auf *umundo.core* und *umundo.s11n* bietet diese Komponente einfache Remote Procedure Calls (RPC-Calls).
  RPC-Calls können genutzt werden, um Prozeduren/Funktionen aufzurufen, die in anderen Prozessen (oder sogar auf anderen Rechnern) laufen. Die angebotenen RPC-Services können via Pattern-Matching gefiltert und dann für RPC-Calls benutzt werden. Unterstützt durch die Discovery von uMundo finden die RPC-Calls automatisch das ausführende Ziel. Leider unterstützt diese Komponente nur die Programmiersprachen C++, Java und Objective-C.

2.4 Die uMundo High-Level API

Die High-Level API von uMundo ist nicht nur relativ einfach zu benutzen, sondern auch durch viele kleinere Beispiele dokumentiert. Im Github Repository von uMundo findet man einige dieser Verwendungsbeispiele[3]. Erwähnenswert sind dabei folgende Dateien:

- **umundo-pingpong.cpp**
  Dies ist ein einfaches Beispiel für einen Subscriber und Publisher im selben Node. Es ist im Prinzip eine Vereinigung der beiden Listings unter „umundo.core“. Zu beachten ist jedoch, dass Subscriber und Publisher nur miteinander kommunizieren können, wenn sie *verschiedenen* Nodes zugeordnet sind. Dieses Beispiel muss also *zweimal* gestartet werden, damit eine Kommunikation zu sehen ist.

[2] Siehe Github Repository unter `https://github.com/google/protobuf/`
[3] Zu finden unter `https://github.com/tklab-tud/umundo/tree/master/apps/tools`

- **umundo-rtp-pub.cpp** und **umundo-rtp-sub.cpp**
  Dieses einfache Beispiel zeigt das RTP unicast Streaming via uMundo. Es funktioniert ähnlich zu umundo-pingpong, ist aber in zwei verschiedene Dateien aufgeteilt (einmal Publisher und einmal Subscriber).

- **umundo-kinect-pub.cpp** und **umundo-kinect-sub.cpp**
  Dieses (etwas größere) Beispiel zeigt, wie man mittels uMundo RTP Streaming das Bild einer Kinect Tiefenkamera streamen kann. Das Streaming kann entweder via Multicast oder via Unicast erfolgen.

- **umundo-phone.cpp** oder **umundo-multicast-phone-pub.cpp** und **umundo-multicast-phone-sub.cpp**
  Dieses Beispiel demonstriert ebenfalls das Streaming von Daten via RTP. Gestreamt werden Audiodaten, die über das Mikrofon der Soundkarte aufgenommen werden. Demonstriert wird auch hier das Streaming sowohl via Unicast (umundo-phone.cpp) als auch via Multicast (umundo-multicast-phone-pub.cpp und umundo-multicast-phone-sub.cpp).

- **umundo-monitor.cpp**
  Dieses Tool dient dem einfachen Beobachten der uMundo Interna. Der Code kann aber auch genutzt werden, um die Verwendung von uMundo besser zu verstehen.

2.4.1 umundo.core: Einfaches Publish/Subscribe via Messages

Einfaches Publish/Subscribe via Messages lässt sich durch nur wenige Zeilen Code bewerkstelligen:

Der Publisher:[4]
Für das Publizieren müssen erst einmal eine Discovery und ein Kommunikationsknoten erstellt werden (siehe Listing 2.4.1).

```
1  //create discovery (mdns) and add a new communication node to it
2  Discovery disc(Discovery::MDNS);
3  Node node;
4  disc.add(node);
5
6  //create publisher for channel "mychannel" and add it to the node
7  Publisher pubFoo("mychannel");
8  node.addPublisher(pubFoo);
9
10 //send message containing "pingexample"
11 Message* msg = new Message();
12 msg->setData("pingexample", 11);
13 pubFoo.send(msg);
14 delete(msg);
```

Listing 2.4.1: uMundo API für simple Publish/Subscribe Anwendungen - Publisher

[4] Der vollständige Code kann im Listing A.1.1 gefunden werden

Der Subscriber:[5]
Um an die Daten zu kommen gibt es zwei Möglichkeiten: Entweder muss mit Polling gearbeitet werden (siehe Listing 2.4.3), oder es wird eine Callback-Klasse benötigt (siehe Listing 2.4.2).

```
1 //simple callback class which receives incoming messages
2 class TestReceiver : public Receiver {
3 public:
4     TestReceiver() {};
5     void receive(Message* msg) {
6         std::cout << msg->data() << std::flush;
7     }
8 };
```

Listing 2.4.2: uMundo API für simple Publish/Subscribe Anwendungen - Subscriber (Callback-Klasse)

```
1 while(sub->hasNextMsg()) {
2     Message* msg = pub->getNextMsg();
3 }
```

Listing 2.4.3: uMundo API für simple Publish/Subscribe Anwendungen - Subscriber (Polling)

Für beide Verfahren müssen erst einmal eine Discovery und ein Kommunikationsknoten erstellt werden (siehe Listing 2.4.4).

```
1  //create discovery (mdns) and add a new communication node to it
2  Discovery disc(Discovery::MDNS);
3  Node node;
4  disc.add(node);
5
6  //create a new callback object (receiver)
7  TestReceiver* testRecv = new TestReceiver();
8
9  //create subscriber for channel "mychannel" and add it to the node
10 //use testRecv callback object (leave out second argument for polling)
11 Subscriber subFoo("mychannel", testRecv);
12 node.addSubscriber(subFoo);
13
14 //the receiver callback will be called from another thread
15 //we don't have to do anything more in this thread
16 while(1)
17     Thread::sleepMs(1000);
```

Listing 2.4.4: uMundo API für simple Publish/Subscribe Anwendungen - Subscriber (Initialisierung)

[5] Der vollständige Code kann im Listing A.1.2 gefunden werden

2.4.2 umundo.core: Einfaches Streaming via RTP

Wie bereits erwähnt, unterstützt uMundo zusätzlich zum klassischen Publish/Subscribe auch eine Variante, die für das Streaming optimiert ist. Im Vergleich zur klassischen Variante ist dafür kaum eine Änderung des Codes nötig.

Will man mehrere Subscriber auf dem gleichen Channel dem Streaming lauschen lassen, so bietet es sich an, Multicast zu verwenden. Denn durch die Verwendung von Multicast wird das Netzwerk viel weniger belastet. Gerade für das Streaming von High Quality Videos an eine große Anzahl an Subscribern ist das interessant.

Zur Konfiguration von Multicast Streaming muss am Publisher nichts geändert werden. Ein Publisher kann sogar sowohl Unicast als auch Multicast Subscriber bedienen (siehe Listing 2.4.5).

Die folgenden Listings 2.4.6 und 2.4.7 verdeutlichen dies. Vollständige Codebeispiele für RTP Streaming können in Listings A.2.1 bis A.2.3 gefunden werden.

```
1 [...]
2 //PCMU data (RTP payload type 0) with sample rate of 8000Hz and 20ms payload↩
      per RTP packet (-> 166 samples)
3 RTPPublisherConfig pubConfig(166, 0);   //additional 3rd argument: network ↩
     port
4 Publisher pubFoo(Publisher::RTP, "mysoundchannel", &pubConfig);
5 [...]
```

Listing 2.4.5: uMundo API für simple RTP Publish/Subscribe Anwendungen - Publisher

```
1 [...]
2 RTPSubscriberConfig subConfig;              //automatic network port ↩
     selection
3 //OR:
4 RTPSubscriberConfig subConfig(42042);       //set network port for unicast ↩
     communication
5 Subscriber subFoo(Subscriber::RTP, "mysoundchannel", &testRecv, &subConfig);
6 [...]
```

Listing 2.4.6: uMundo API für simple RTP Publish/Subscribe Anwendungen - Unicast Subscriber

```
1  [...]
2  RTPSubscriberConfig subConfig;                //automatic network port ↩
       selection
3  subConfig.setMulticastPortbase(42042);
4  subConfig.setMulticastIP("239.1.2.3");       //not needed (default multicast ↩
       group: 239.8.4.8)
5  Subscriber subFoo(Subscriber::RTP, "mysoundchannel", &testRecv, &subConfig);
6  [...]
```

Listing 2.4.7: uMundo API für simple RTP Publish/Subscribe Anwendungen - Multicast Subscriber

2.5 VPN

Ein Virtual Private Network (VPN) ist ein virtuelles Netzwerk, dessen Daten (meist verschlüsselt) über ein darunter liegendes Netzwerk zwischen den Teilnehmern übertragen werden. Ein Teilnehmer kann hierbei entweder ein einzelner Rechner oder auch ein ganzes Netzwerk sein. Ziel eines solchen VPNs ist es, ein logisches Subnetz zu bilden, also die Teilnehmer so zu verbinden, als seien sie direkt verkabelt. Meist ist die Verbindung zwischen den Teilnehmern verschlüsselt, da die Daten über ein unsicheres Netzwerk wie beispielsweise das Internet laufen. Die bekanntesten und am weitesten verbreiteten VPN Lösungen sind:

- **OpenVPN [opea]**
  OpenVPN steht unter der GPL [gpl07] und läuft auf allen Plattformen, auf denen uMundo auch benutzt werden kann.

- **Internet Protocol Security (IPsec) [KS05]**
  Von IPsec existieren verschiedene (manchmal auch inkompatible) Implementierungen unter verschiedenen Lizenzen. Für alle Plattformen, auf denen uMundo läuft, existieren aber auch Implementierungen von IPsec. **IPsec unterstützt allerdings kein Multicast, kann also für die Verbindung zweier uMundo Workspaces nicht verwendet werden.**

Abbildung 2.5.1 zeigt die verschiedenen Verwendungsmöglichkeiten eines VPNs. Im Anhang finden sich außerdem die Listings A.3.1 und A.3.2, die die Konfiguration eines OpenVPN basierten VPNs zeigen, bei dem zwei einzelne Rechner verbunden werden (Peer-to-Peer Modus).
Der Unterschied zwischen der Konfiguration des Servers und Clients besteht in diesem Beispiel nur aufgrund des Unterschieds in der darunter liegenden Netzwerktopologie bzw. Netzwerktechnik. Die Firewall des Clients muss ausgehende Verbindungen zu UDP Port 1194 erlauben, die Firewall des Servers eingehende UDP Verbindungen auf Port 1194.
Abgesehen von der Konfiguration der Firewall oder des DSL-Routers besteht aber kein Unterschied zwischen VPN Server und Client und die Rollen der beiden beteiligten Rechner können auch problemlos vertauscht werden.

Literatur zu OpenVPN kann unter [Bec11, Zel08] oder auch online unter [opeb] und [wik] gefunden werden.

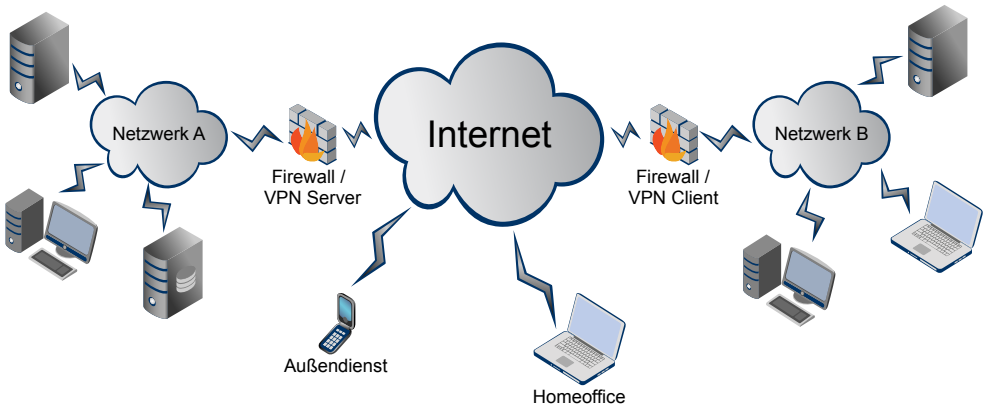


Abbildung 2.5.1.: Die Verwendung eines VPNs zur sicheren Verbindung mehrerer Teilnehmer über das Internet

3 Related Work

Neben uMundo existieren noch weitere Publish/Subscribe Frameworks und auch andere Kommunikationslösungen, die alternativ genutzt werden können.

3.1 Synapse

Synapse [syn] ist ein „Advanced event framework in C++“, welches unter GPLv3 [gpl07] steht und modular aufgebaut ist. Es folgt laut Dokumentation dem Publish/Subscribe Pattern und lässt sich daher (wie uMundo auch) zu den Publish/Subscribe Frameworks zählen. Auf der Webseite des Projekts[6] sind folgende Besonderheiten von Synapse genannt:

- **Modular plugin system**
  Es können eigene Module zum Senden/Empfangen von Daten über ein beliebiges Transportmedium geschrieben werden.

- **Easy to write modules**
  Einzelne Module können relativ einfach programmiert und in das bestehende System integriert werden.

- **Easy to send and receive message, while maintaining code readability**
  Die Benutzung des Systems und das Senden/Empfangen von Publish/Subscribe Nachrichten über Synapse ist mit wenigen Codezeilen machbar.

- **Session management**
  Eine Session entspricht einem *Channel* oder *Topic* in anderen Publish/Subscribe Systemen.

- **Build in authentication**
  Die Authentifizierung kann genutzt werden, um einzuschränken, welche Publisher/Subscriber überhaupt miteinander kommunizieren dürfen. Authentifizierung findet bei „klassischem“ Publish/Subscribe nicht statt.

- **Fine grained access control on sending and receiving of events**
  Events können, wie bei Publish/Subscribe generell üblich, gefiltert werden.

- **Events can be handled by multiple threads if desired (max-threads is adjustable per session and per module)**
  Ankommende Events/Daten können in mehreren Threads gleichzeitig bearbeitet werden. Ein explizites Locking ist dafür nicht nötig.

[6] Zu finden unter `http://open.syn3.nl/syn3/trac/default/wiki/projects/synapse`

Synapse bietet damit eine ähnliche Kommunikation wie uMundo, verfolgt allerdings ein etwas anderes Konzept und ist auch nicht für das Streaming von Daten via Real-Time Transport Protocol (RTP) [SCFJ03] optimiert.
Auch besitzt Synapse keine Discovery, die beteiligten Kommunikationspartner müssen sich also explizit direkt miteinander verbinden.

Synapse ist für die Verwendung in einem Netzwerk gedacht und bietet neben „normaler“ TCP/IP basierter Kommunikation auch ein fertiges Modul, um via Hypertext Transfer Protocol (HTTP) [FGM+99] und Javascript Daten zu senden und zu empfangen. Intern werden die Daten immer in JavaScript Object Notation (JSON) [Bra14] übertragen.

3.2 OMG Data-Distribution Service for Real-Time Systems

Der Data-Distribution Service for Real-Time Systems (DDS) der Object Management Group (OMG) beschreibt sich selbst als den „ersten offenen internationalen Middleware-Standard, der direkt Publish/Subscribe Kommunikation für real-time und eingebettete Systeme adressiert“ [dds]. Er bietet neben einfachem Publish/Subscribe auch erweiterte Funktionalitäten wie Quality of Service (QoS). Es werden hierbei die QoS Merkmale *Reliability*, *Bandwidth*, *Delivery Deadlines* und *Resource Limits* unterstützt. Die Daten selbst werden bei DDS unter *Topics* veröffentlicht, die weitestgehend den *Channels* von uMundo entsprechen.

Verwendet werden kann DDS beispielsweise über The Simple DDS API (SimD)[7] von Prismtech[8] für C++, welche unter der LGPL [lgp07] steht. Es existieren aber auch noch weitere Implementierungen des DDS Standards für verschiedene andere Plattformen und Programmiersprachen.

Verwendung findet DDS in vielen großen Firmen (beispielsweise bei Microsoft). Vom Funktionsumfang und der Art der Bedienung kommt die DDS Spezifikation recht nahe an uMundo heran bzw. übertrifft uMundo in manchen Punkten sogar.

Im Gegensatz zu uMundo ist DDS aber zuerst einmal nur eine Spezifikation, die noch programmtechnisch umgesetzt werden muss. Das bedeutet, dass es für DDS viele verschiedene Implementierungen für verschiedene Programmiersprachen und Plattformen gibt. Diese sind teilweise inkompatibel zueinander und stehen darüber hinaus auch noch unter verschiedenen Lizenzen. Will man ein Framework verwenden, welches nicht nur viele Plattformen und Programmiersprachen unterstützt, sondern auch unter einer Lizenz steht, die nicht nur das Verwenden in Open Source Produkten, sondern auch die Veränderung und Verwendung in Closed Source Produkten erlaubt, so ist DDS trotz all seiner Funktionen nicht die richtige Wahl.

Im Vergleich zu uMundo, welches unter der 3-Clause BSD Lizenz steht und damit sehr frei verwendet werden kann, sind die meisten DDS Frameworks unter der (L)GPL lizenziert und damit lange nicht so frei verwendbar wie uMundo. Auch unterstützt DDS kein echtes Streaming

[7] Zu finden unter `https://code.google.com/p/simd-cxx/`
[8] Die Webseite der Firma ist unter `http://www.prismtech.com/` zu finden

über RTP und die meisten Frameworks, die DDS umsetzen, sind nicht für die Verwendung unter iOS oder Android optimiert.

Alles in allem sind DDS und die verschiedenen Frameworks, die diesen Standard umsetzen, eine gute Alternative zu uMundo, können aber in Flexibilität im Hinblick auf die unterstützten Programmiersprachen, Plattformen und die Lizenz nicht mit uMundo mithalten.

3.3 ZeroMQ

ØMQ[9] (auch ZeroMQ, 0MQ oder ZMQ genannt und im Folgenden der Lesbarkeit wegen „ZeroMQ“) ist eine leichtgewichtige Netzwerkbibliothek, die asynchrone Input/Output (I/O) Operationen ermöglicht und unter LGPL [lgp07] steht.
ZeroMQ besitzt eine sehr gute Dokumentation [Hin] und ist einfach zu handhaben. Die ZeroMQ API bietet dem Programmierer Sockets, die über viele verschiedene Transportprotokolle bzw. Transportwege arbeiten, je nachdem, was der Programmierer benötigt.
Unterstützt werden in-process Kommunikation, inter-process Kommunikation, die Kommunikation über TCP und die Verwendung von Multicast. Zusätzlich zu den möglichen Transportwegen bietet ZeroMQ die Auswahl von verschiedenen Kommunikationspatterns beim Erzeugen eines Sockets. Diese Patterns werden von ZeroMQ automatisch durchgesetzt, ein Verhalten außerhalb des ausgewählten Pattern ist also nicht möglich oder wird zumindest erschwert.
Das *request-reply* Pattern beispielsweise erfordert zwingend, dass immer abwechselnd **zmq_recv**() und **zmq_send**() aufgerufen wird. Das doppelte Aufrufen von **zmq_recv**() führt zu einem Fehler/einer Exception (je nach verwendeter Programmiersprache). Die von ZeroMQ unterstützten Patterns sind: *fan-out*, *pub-sub*, *task distribution* und das bereits erwähnte *request-reply*.
Verschickt werden die Daten dabei immer als ZeroMQ Message und der Empfang der Daten erfolgt immer in atomaren Blöcken (den Messages).

Neben den vielen verschiedenen Transportwegen und Kommunikationspatterns, die ZeroMQ anbietet, ist auch die Liste der unterstützten Plattformen und Programmiersprachen lang. Eine (vermutlich nicht erschöpfende) Liste von unterstützten Programmiersprachen ist C, C++, C#, CL, Delphi, Erlang, F#, Felix, Go, Haskell, Java, Lua, Node.js, Objective-C, Perl, PHP, Python, Q, Ruby, Scala, Tcl, Ada, Basic, Clojure, Haxe, ooc und Racket.
Unterstützt werden laut Webseite „alle modernen Plattformen“, wozu auch Android, iOS, Mac OS X, Windows und Linux zählen.

Da ZeroMQ als Kommunikationspattern auch Publish/Subscribe anbietet, kann es natürlich auch als Unterbau eines größeren Publish/Subscribe Frameworks verwendet werden. Konsequenterweise wird ZeroMQ als Unterbau von uMundo verwendet. Dadurch wird jede Kommunikation in uMundo (mit Außnahme der mDNS Discovery) über ZeroMQ abgewickelt. Plain TCP- oder UDP-Sockets werden nicht benutzt.

ZeroMQ findet unter anderem Verwendung in folgenden Firmen: AT&T, Cisco, EA, Los Alamos Labs, NASA, Weta Digital, Zynga, Spotify, Samsung Electronics, Microsoft und am CERN[10].

[9] Die Webseite des Projekts findet sich unter `http://zeromq.org/`
[10] Siehe auch `http://zeromq.org/intro:read-the-manual#toc3`

3.4 Client/Server als Alternative zu Publish/Subscribe

Neben den verschiedenen Publish/Subscribe Lösungen lassen sich auch noch andere Alternativen für die Kommunikation zwischen den Endgeräten nutzen.
Ein Beispiel wäre eine einfache Client/Server Architektur, bei der die einzelnen (smarten) Endgeräte über eine standardisierte Representational State Transfer (REST) Schnittstelle oder Remote Procedure Call (RPC) kommunizieren (bzw. sich steuern lassen). Auch hierfür können sich die Endgeräte mit mDNS (oder „normalem“ DNS) und darauf aufsetzend DNS-SD finden und sich so konfigurieren, dass eine automatische Kommunikation ermöglicht wird.

3.5 Virtual Private Network

Ein Virtual Private Network (VPN) ist als Erweiterung für eine einfache Client/Server Lösung denkbar und würde diese Lösung um Verschlüsselung und Authentifizierung erweitern.
Aber auch andere Kommunikationsarten lassen sich mit einem VPN um diese Funktionen erweitern. In Abschnitt 2.5 wurde die generelle Verwendung eines VPNs bereits genauer beschrieben.

4 Konzepte

Um verschiedene uMundo Nodes über Routergrenzen hinweg zu verbinden, wird eine Brücke benötigt. Die folgenden Konzepte verdeutlichen die Anforderungen an eine solche Brücke und zeigen die verschiedenen Implementierungsmöglichkeiten auf.

4.1 Anforderungen

Die Brücke sollte folgende Kriterien erfüllen:

1. Verbindung zweier ganzer Subnetze über Routergrenzen hinweg möglich
2. Verbindung eines einzelnen Gerätes mit einem ganzen Subnetz über Routergrenzen hinweg möglich
3. Verbindung eines einzelnen Gerätes mit einem ganzen Subnetz über Routergrenzen hinweg möglich
4. Flüssige Datenübertragung, benötigte Bandbreite nicht signifikant höher als die lokal benötigten Bandbreite
5. *Optional:* Sichere Übertragung der Daten (Verschlüsselung, Authentifizierung etc.)
6. Möglichst einfache Einrichtung (während die Sicherheit weiterhin gewährleistet bleibt, falls gewünscht)

4.2 VPN

Wie die Beschreibung eines Virtual Private Networks (VPNs) in Abschnitt 2.5 zeigt, lassen sich mit einem VPN sowohl ganze vorher getrennte Subnetze zu einem neuen logischen Subnetz zusammenfassen als auch einzelne Netzwerkteilnehmer in ein solches logisches Subnetz einbinden. Das in Kapitel 1 beschriebene Problem der Kommunikation über Netzwerkgrenzen hinweg lässt sich also durch die Verwendung eines VPNs lösen. Damit funktioniert die automatische Discovery der uMundo Knoten via mDNS wieder, obwohl sich die kommunizierenden Knoten in verschiedenen physikalischen Netzsegmenten befinden.
Am Beispiel von OpenVPN [opea] soll ein solches VPN nun näher auf seine Tauglichkeit für die Verbindung mehrerer uMundo Workspaces untersucht werden.

Das Setup für die Verbindung zweier Firmenstandorte sieht damit wie folgt aus: Die beiden physikalischen Subnetze an Standort A und Standort B müssen so konfiguriert werden, dass sie identisch sind (etwa 10.13.0.0/16). An beiden Standorten muss ein VPN-Server eingerichtet

werden. Diese beiden VPN-Server bilden dann die Brücke zwischen den beiden Netzwerken. Damit diese Brücke auch tatsächlich funktioniert, müssen noch die virtuellen Netzwerkkarten auf den VPN-Servern mit den physikalischen Netzwerkkarten gebridget werden. Diese Konfiguration sorgt dafür, dass Ethernet-Frames (Open Systems Interconnection Model (OSI) Layer 2 [osi94]) zwischen den beiden Standorten ausgetauscht werden können. Dazu zählen auch Broadcast Pakete oder Multicast Pakete, wie sie für mDNS genutzt werden. Beide Netzwerke dürfen nach ihrer Verbindung nur noch einen einzigen DHCP-Server benutzen, da sonst IP-Adresskonflikte erzeugt werden.

Die virtuellen Netzwerkkarten müssen zwingend auf OSI Layer 2 [osi94] arbeiten (TAP-Modus) und mit den physikalischen Netzwerkkarten gebridget werden. Eine Verbindung der beiden Firmenstandorte über virtuelle Netzwerkkarten im TUN-Modus (OSI Layer 3 [osi94]) erfordert Routing und führt damit wieder eine Routergrenze zwischen den beiden Standorten ein, die von Multicast Paketen nicht überquert werden kann.

Das hier skizzierte Minimalsetup ist bereits relativ kompliziert, kann aber durch Besonderheiten in der Netzwerktopologie noch komplexer werden. Punkt 6 der obigen Anforderungsliste (möglichst einfache Einrichtung) ist damit leider nicht erfüllt.

Die Verbindung eines einzelnen Rechners (etwa im Homeoffice) mit dem Firmenstandort ist etwas weniger komplex, da hier die Konfiguration auf dem Rechner einfacher ausfällt. Hier muss kein Verbinden von physikalischen und virtuellen Netzwerkkarten stattfinden und auch kein zweiter DHCP-Server berücksichtigt werden. Der Konfigurationsaufwand auf der Serverseite besteht aber weiterhin.

Überträgt man das Beispiel der Verbindung mit dem Firmenstandort auf die Verbindung eines Laptops mit dem Heimnetzwerk, so kommen sogar noch weitere Probleme hinzu: Die meisten Heimnetzwerke sind auf das gleiche Subnetz konfiguriert (Beispielsweise 192.168.178.0/24 bei Routern der FritzBox Familie der Firma AVM). Die virtuelle Netzwerkkarte benötigt aber zwingend eine IP aus dem Heimnetzwerk, zu dem der Laptop sich verbinden soll. Befindet sich der Laptop in einem anderen Heimnetz (etwa in dem eines Freundes) und soll sich mit dem eigenen Heimnetz verbinden, so bekommen die virtuelle und die physikalische Netzwerkkarte auf dem Laptop deshalb in vielen Fällen IP-Adressen aus dem gleichen Subnetz zugeteilt. Das Resultat ist eine eingeschränkte oder nicht vorhandene Konnektivität aufgrund von Routingproblemen. Lösen lässt sich dieses Problem bis zu einem gewissen Grad nur, indem im eigenen Heimnetzwerk ein selten verwendetes Subnetz gewählt wird (etwa 10.13.77.0/24).

Es besteht allerdings noch ein weiterer Nachteil: Werden zwei Netzwerke auf diese Weise verbunden, so wird nicht nur die uMundo-Kommunikation durch das VPN geleitet, sondern jegliche Kommunikation. Soll nur die Kommunikation via uMundo möglich sein und jegliche andere Kommunikation zwischen den Netzwerken unterbunden werden, so ist die Einrichtung einer dafür nötigen Firewall ziemlich kompliziert bzw. fast unmöglich. Für eine komplette Abschottung der Netzwerke gegeneinander müsste die Firewall DNS-SD und mDNS Pakete blockieren, die nicht von uMundo kommen. Außerdem müsste sie aus der durchgelassenen mDNS Kommunikation schließen, auf welchen Ports die uMundo Nodes miteinander kommunizieren wollen. Diese TCP und UDP Ports müsste die Firewall dann automatisch zu Beginn der Kommunikation

öffnen und danach wieder schließen. Sie müsste also uMundo „verstehen“ und wäre damit fast schon selbst eine Art Bridge, wie sie in Abschnitt 4.3 beschrieben wird.

Eine solche OpenVPN basierte Verbindung zweier uMundo Workspaces benötigt kaum eine größere Bandbreite als die direkte Kommunikation. Mit der OpenVPN Option „comp-lzo“ lässt sich die benötigte Bandbreite durch Kompression sogar noch weiter reduzieren.

Zusammenfassend kann man sagen, dass ein solches VPN zwar uMundo um Verschlüsselung und Authentifizierung erweitert und auch die Punkte 1 bis 5 der obigen Anforderungsliste erfüllt, Punkt 6 der Anforderungen wird hingegen nicht erfüllt: Die Konfiguration eines für uMundo verwendbaren VPNs ist sehr komplex.

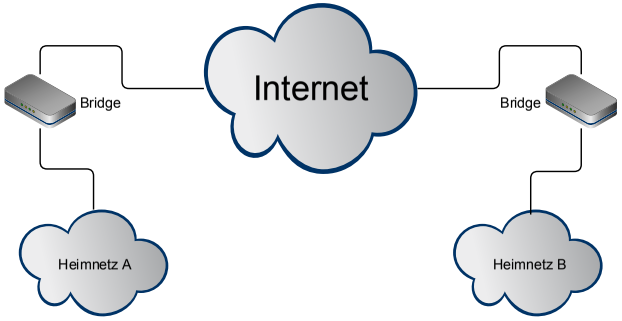
4.3 Bridge

Eine weitere Möglichkeit mehrere uMundo Workspaces zu verbinden ist die Nutzung eines dedizierten uMundo Proxies (im Folgenden nun *Bridge* genannt). Diese Bridge soll die gewünschte Kommunikation sowohl zwischen zwei ganzen Netzwerken (uMundo Workspaces) als auch zwischen einem einzelnen Gerät und einem uMundo Workspace einfach herstellen können. Außerdem sollte eine solche Bridge weniger Nachteile aufweisen als die Lösung mittels eines VPNs.

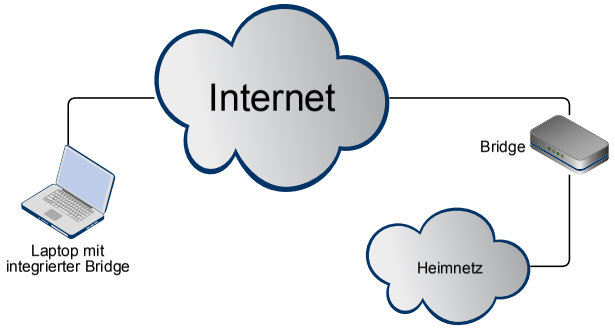
Die Abbildungen 4.3.1a bis 4.3.1c zeigen schematisch, wie eine solche Bridge im Netzwerk verwendet wird.
Sollen mehr als zwei Netzwerke verbunden werden, so darf kein voll vermaschtes Netz zwischen den einzelnen Teilnehmern entstehen, da sonst ein geschlossener Ring gebildet wird, in dem die Daten immer im Kreis wandern. Stattdessen sollte eine sternförmige Topologie gewählt werden, bei der sich alle Teilnehmer mit einem zentralen Bridge-Server verbinden. Dieser Server muss natürlich über eine ausreichend breitbandige Anbindung verfügen, um die einzelnen Brücken zu versorgen. Ein solches Setup wird in Abbildung 4.3.1d skizziert.

Um eine Brücke aufzubauen, reicht es, auf den beiden Endpunkten das entsprechende Programm *umundo-bridge* einmal als Server und einmal als Client zu starten. Die möglicherweise auf dem Server vorhandene Firewall muss eingehende Verbindungen auf dem konfigurierten Port für die Protokolle TCP und UDP erlauben. Die Listings 4.3.1 und 4.3.2 zeigen beispielhaft den Aufruf des Programms als Server und als Client. Ist die Verbindung wie in diesem Beispiel erfolgreich, so können auf beiden Seiten uMundo-Programme gestartet werden und sich dann über die Bridge miteinander verbinden. Normale Kommunikation wird von der Bridge dabei über TCP übermittelt, für das Streaming optimierte Kommunikation über UDP. Ist nur eine Kommunikation über UDP möglich, so leitet die Bridge nur normale uMundo-Kommunikation weiter und ignoriert die via RTP gestreamten Inhalte. Näheres dazu in Kapitel 5.

Damit die Kommunikation zwischen den beiden uMundo Workspaces funktioniert, erzeugt die Bridge stellvertretend für alle Publisher im lokalen Workspace ebenfalls Publisher im entfernten Workspace. Subscriptions auf diesen stellvertretenden Publishern sorgen dann dafür, dass die Bridge stellvertretend für den entfernten Subscriber im lokalen Workspace ebenfalls auf dem entsprechenden Channel subscribt. Veröffentlicht ein Publisher im lokalen Workspace Daten,

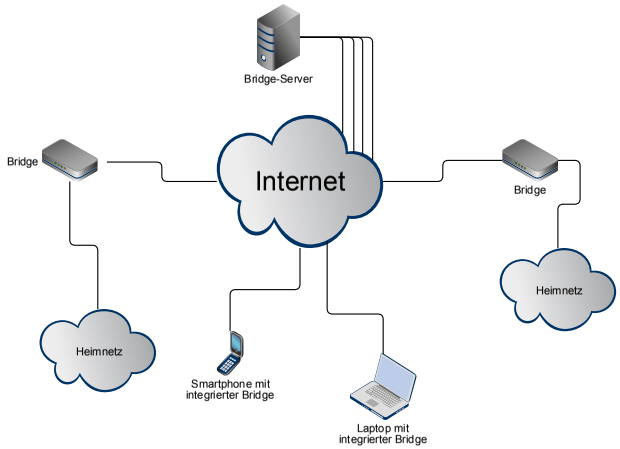


(a) Verbindung zweier uMundo Workspaces



(b) Verbindung eines einzelnen Gerätes zu einem uMundo Workspace

(c) Verbindung zweier einzelner Geräte zu einem neuen (virtuellen) uMundo Workspace



(d) Verbindung mehrerer uMundo Workspaces und Geräte über einen zentralen Bridge-Server

Abbildung 4.3.1.: Die verschiedenen Verwendungsarten der uMundo Bridge zum Verbinden von uMundo Geräten

so werden diese über den stellvertretenden Subscriber empfangen und am entfernten Endpunkt der Bridge durch den stellvertretenden Publisher wieder veröffentlicht. Der entfernte Subscriber bekommt so die Daten des lokalen Publishers. Die Bridge ist bidirektional: Entfernte Publisher werden ebenfalls durch Stellvertreter im lokalen Workspace und die Subscriber im lokalen Workspace durch Stellvertreter im entfernten Workspace emuliert. Abbildung 4.3.2 verdeutlicht diesen Vorgang grafisch.

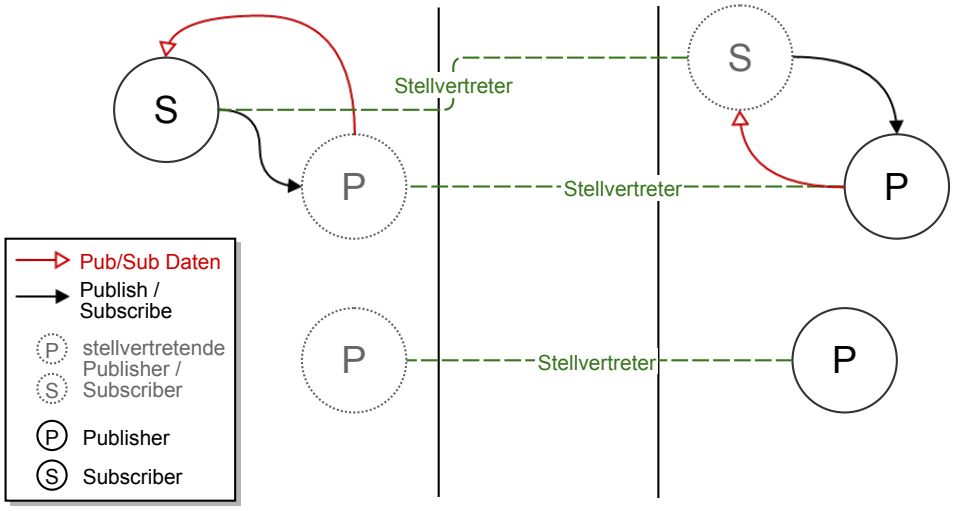


Abbildung 4.3.2.: Publish/Subscribe Kommunikation über die Bridge. Die Weiterleitung der Daten geschieht über Stellvertreter an den Bridge Endpunkten.

In eine solche Bridge ließen sich zwar auch Verschlüsselung und Authentifizierung einbauen, allerdings wäre der Code der Bridge dann komplizierter. Durch das Einbauen von Sicherheitsfeatures, wie Verschlüsselung und Authentifizierung, würde uMundo nicht mehr so leichtgewichtig sein wie aktuell. Wären solche sicherheitsrelevanten Teile direkt in uMundo und die Bridge integriert, so müssten diese regelmäßig gewartet werden, um möglicherweise gefundene Sicherheitslücken zu schließen. Eine solche Wartung ist aber nicht nur zeitintensiv, sondern sollte auch echten Sicherheitsexperten überlassen werden. Eine nicht regelmäßig durchgeführte Wartung würde darüber hinaus den Nutzern der Bridge eine Sicherheit vorgaukeln, die effektiv durch das Vorhandensein von möglichen Sicherheitslücken gar nicht gegeben ist. Wendet man hingegen externe Sicherheitsfunktionen auf die Bridge an (siehe Abschnitt 4.4), so können diese durch automatische Systemupdates immer auf dem neuesten Stand gehalten werden und müssen nicht durch die Entwickler von uMundo selbst gewartet werden. Da das benötigte Sicherheitslevel auch vom Anwendungsfall abhängt, wäre eine fest in die Bridge eingebaute Verschlüsselung/Authentifizierung nicht so flexibel wie eine von der Bridge losgelöste Variante. Aus diesen Gründen ist die im Rahmen dieser Bachelorarbeit entwickelte Bridge nicht verschlüsselt und bietet keine Möglichkeit der Authentifizierung.

Zusammenfassend kann man sagen, dass die hier skizzierte Bridge die unter Abschnitt 4.1 genannten Anforderungen fast alle erfüllt. Nur der optionale Punkt 5 der Anforderungen (Verschlüsselung und Authentifizierung) wird nicht erfüllt.
Dafür ist die Handhabung aber um einiges einfacher als bei einem VPN.
Die zusätzliche Anforderung, dass die Bridge weniger Nachteile aufweisen sollte als die Lösung

mittels eines VPNs, ist ebenfalls erfüllt. Auch die für die Bridge verwendete Bandbreite ist in der gleichen Größenordnung wie die Bandbreite, die ein VPN benötigen würde.

```
# bin/umundo-bridge -v -l 5555
umundo-bridge version 0.4.4 (Debug build)
Listening at tcp and udp port 5555...
Accepted connection from 127.0.0.1:34917 on tcp socket, sending ↩
   serverHello message...
Connection successfully established, now handing off to ↩
   ProtocolHandler...
[...]
INFO: sending internal ping message on UDP channel...
INFO: sending internal ping message on TCP channel...
INFO: received internal ping message using UDPListener...
INFO: received internal ping message using TCPListener...
[...]
Terminating TCPListener...
ProtocolHandler lost connection...
Shutting down ProtocolHandler...
Terminating UDPListener...
Listening for new connections in 5 seconds...
Listening at tcp and udp port 5555...
[...]
```

Listing 4.3.1: Aufruf der Bridge als Server

```
# bin/umundo-bridge -v -c localhost:5555
umundo-bridge version 0.4.4 (Debug build)
Connecting to remote bridge instance at localhost:5555
Info: 'localhost' is at 127.0.0.1...
Connection successfully established, now handing off to ↩
   ProtocolHandler...
[...]
INFO: sending internal ping message on UDP channel...
INFO: sending internal ping message on TCP channel...
INFO: received internal ping message using UDPListener...
INFO: received internal ping message using TCPListener...
[...]
Terminating TCPListener...
ProtocolHandler lost connection...
Shutting down ProtocolHandler...
Terminating UDPListener...
Reconnecting in 5 seconds...
Connecting to remote bridge instance at localhost:5555
[...]
```

Listing 4.3.2: Aufruf der Bridge als Client

4.4 Hybrid: Bridge + VPN

Wird, wie in Abschnitt 4.3 beschrieben, eine Bridge verwendet, so verliert man die Verschlüsselung und Authentifizierung, die die Verwendung eines VPNs wie in Abschnitt 4.2 bieten würde. Andererseits sorgt die Verwendung der Bridge aber dafür, dass für die Kommunikation zwischen den Endpunkten der Brücke nur noch TCP und UDP verwendet werden und die Unterstützung von Multicast auf dieser Strecke nicht mehr notwendig ist.

Die beiden Konzepte *Bridge* und *VPN* ergänzen sich sehr gut und können deshalb gemeinsam angewendet werden. Die daraus resultierende Lösung bietet neben der einfachen Einrichtung und Nutzung der in Abschnitt 4.3 skizzierten Bridge auch noch Verschlüsselung und Authentifizierung. Die Unterstützung von Multicast ist hier keine notwendige Anforderung an das VPN mehr; die Einrichtung eines solchen wird daher auch noch um einiges einfacher.

Die Sicherheit (Verschlüsselung, Authentifizierung) wird bei diesem hybriden Konzept durch ein Point-to-Point VPN gewährleistet, welches nur 2 IP-Adressen nutzt. Diese beiden IP-Adressen können zufällig gewählt werden und erzeugen nur Adresskonflikte, wenn sie in den IP-Adressbereich von einem der teilnehmenden Rechner fallen.
Verwendet man das VPN nur, um die beiden Bridge-Instanzen darüber kommunizieren zu lassen, so ist es auch nicht mehr zwingend notwendig, den TAP-Modus zu nutzen oder die virtuelle und physikalische Netzwerkkarte zu bridgen. Weil keine Multicast Pakete mehr über das VPN geschickt werden müssen, kann sogar IPsec [KS05] als VPN genutzt werden.

Soll keine RTP-Kommunikation durch die Bridge geleitet werden, so muss auch nur noch die TCP-Kommunikation durch das VPN geschickt werden. Für einen solchen verschlüsselten Tunnel kann dann selbst einfaches SSH genutzt werden. Möglich sind generell aber alle Tunnelmechanismen, die TCP (und für RTP zusätzlich auch UDP) tunneln können.

Zusammenfassend kann man sagen, dass die Verwendung einer Bridge über einen VPN Tunnel einfach einzurichten ist und allen Anforderungen im Abschnitt 4.1 entspricht.

5 Implementierung

In diesem Kapitel wird die Implementierung der in Kapitel 4 skizzierten Bridge näher erläutert. Zuerst wird in Abschnitt 5.1 beschrieben, welche Voraussetzungen uMundo bieten muss, damit eine Bridge implementiert werden kann. Auf die Beschreibung eines ersten Implementierungsversuchs in Abschnitt 5.2, der sich als Sackgasse erwies, folgt dann die Erläuterung der endgültigen Implementierung in Abschnitt 5.3. Ein Verwendungsbeispiel des hybriden Konzepts in Abschnitt 5.4 schließt dieses Kapitel ab.

5.1 Voraussetzungen

Um zwei uMundo Workspaces zu verbinden, muss die Bridge alle Publisher kennen, die in den beiden Workspaces aktiv sind und stellvertretend für diese im jeweils anderen Workspace unter dem gleichen *Channel* publizieren (Announcement Phase). Da die uMundo API zu Beginn dieser Arbeit noch keine API zur Verfügung stellte, über die das Hinzukommen oder Wegfallen eines Publishers im Workspace erkannt werden konnte, wurde die API entsprechend um einen neuen *Monitor* ergänzt.

In uMundo existieren zu jedem *Publisher*, *Subscriber* und *Node* entsprechende Stubs. Die Stubs repräsentieren entfernte Objekte, beispielsweise einen anderen Node oder einen Publisher/Subscriber auf diesem entfernten Node. Auch der neue Monitor bekommt als sein Argument einen Publisher-Stub. Listing 5.1.1 zeigt die Verwendung dieses neuen Monitor-Interfaces und die verfügbaren Callbacks dieses Interfaces. Die Aufrufe der Methoden *added*, *removed* und *changed* erfolgen innerhalb eines uMundo-internen Threads. Da uMundo während des Aufrufs des Callbacks blockiert wird, sollte der dort ausgeführte Code keine lange Laufzeit haben.

```
1  class PubMonitor : public ResultSet<PublisherStub> {
2  public:
3      PubMonitor();
4      //new publisher added, pubStub contains its stub
5      void added(PublisherStub pubStub);
6      //publisher removed, pubStub contains its stub
7      void removed(PublisherStub pubStub);
8      //publisher added or removed, pubStub contains its stub
9      void changed(PublisherStub pubStub);
10 };
11
12 //usage:
13 Node node;
14 PubMonitor monitor;
15 innerNode->addPublisherMonitor(&monitor);
```

Listing 5.1.1: uMundo API Publisher-Monitor

Erfolgt auf einem dieser stellvertretenden Publisher eine Subscription, so wird von der Bridge im Workspace des originalen Publishers ebenfalls ein Subscriber auf dem entsprechenden Channel erzeugt (Subscription Phase). Dieser Subscriber ist stellvertretend für den originalen Subscriber. Danach leitet die Bridge alle vom originalen Publisher publizierten Daten über den stellvertretenden Publisher und Subscriber bis zum originalen Subscriber im entfernten Workspace weiter (Communication Phase).

Die Bridge benutzt das bereits in uMundo vorhandene *Greeter-Interface*, um neue Subscriptions auf den stellvertretenden Publishern zu erkennen.
Dieses Greeter-Interface ist dafür gedacht, dass ein Publisher Buch führen kann, welche Subscriber er besitzt. Für jeden neu hinzukommenden Subscriber wird die Methode **welcome**() mit dem Stub des Subscribers aufgerufen. Jede Unsubscription löst einen Aufruf der Methode **farewell**() aus.

Listing 5.1.2 zeigt die Verwendung eines Greeters und Abbildung 5.1.1 verdeutlicht die erklärten Zusammenhänge grafisch in einem UML-Sequenzdiagramm.

```
1  class GlobalGreeter : public Greeter {
2  public:
3      GlobalGreeter();
4      //new subscriber added, subStub contains its stub, pub is the publisher
5      void welcome(Publisher& pub, const SubscriberStub& subStub);
6      //subscriber vanished, subStub contains its stub, pub is the publisher
7      void farewell(Publisher& pub, const SubscriberStub& subStub);
8  };
9
10 //usage:
11 GlobalGreeter ownGreeter;
12 Publisher pubFoo("mychannel");
13 pubFoo.setGreeter(&ownGreeter);
```

Listing 5.1.2: uMundo API Greeter-Interface

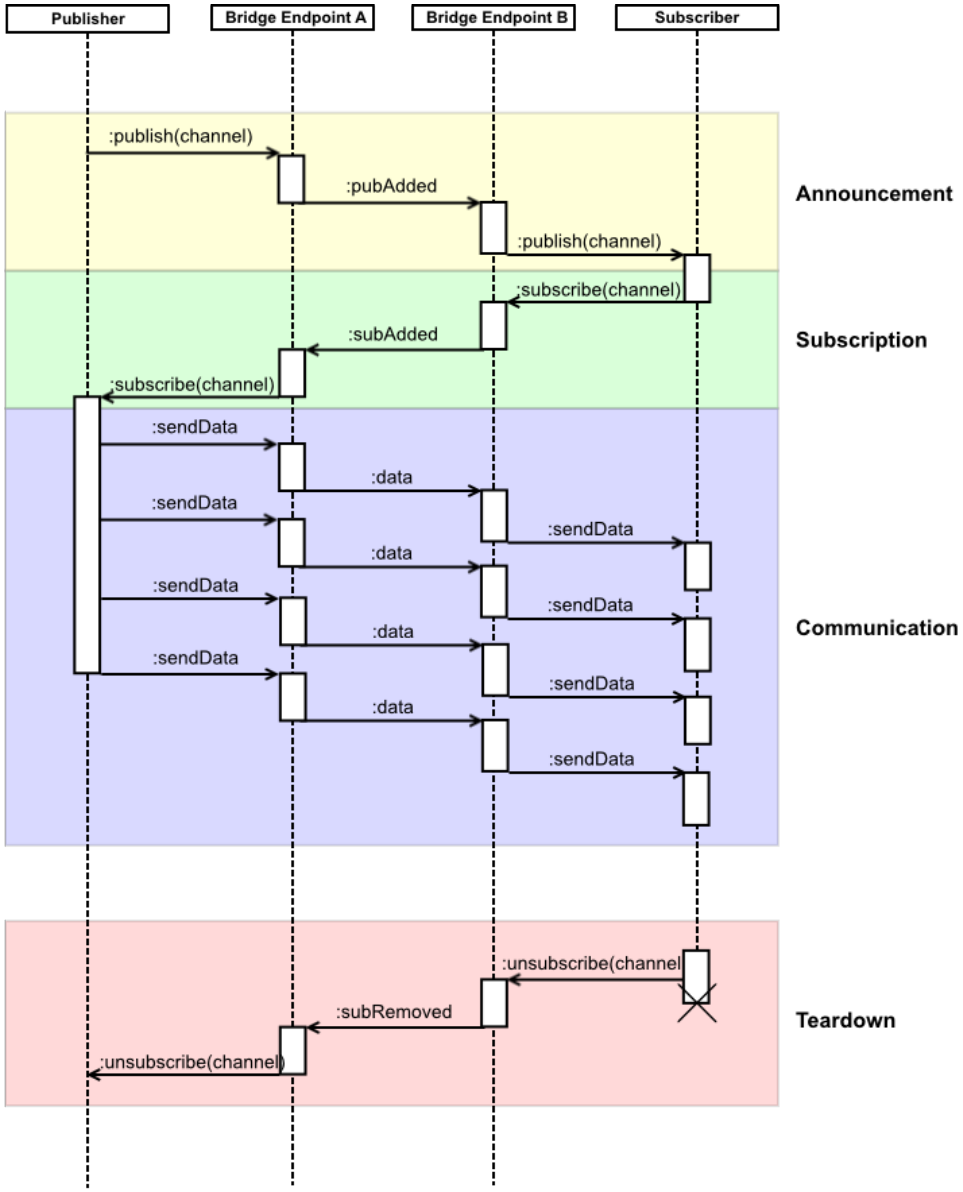


Abbildung 5.1.1.: Sequenzdiagramm der Kommunikation über die Bridge

5.2 Ein erster Implementierungsversuch

Die erste Implementierung der Bridge verwendete uMundo auch für die Kommunikation zwischen den beiden Endpunkten der Bridge. Das hatte den Vorteil, dass die gleiche Implementierung sowohl für die Kommunikation mit den uMundo Workspaces als auch für die Kommunikation zwischen den Endpunkten der Bridge verwendet wurde. Jeder Endpunkt der Bridge hatte dafür zwei uMundo Nodes. Einen Knoten, der via Discovery auf Publisher und Subscriber im lokalen uMundo Workspace lauschte und einen Node, der mittels **node.added(endPoint)**; mit dem entfernten Endpunkt der Bridge verbunden wurde.

Für jeden gefundenen Publisher auf dem einen Node des lokalen Bridge-Endpunktes wurde auf dem jeweils anderen Node ebenfalls ein Publisher erzeugt und umgekehrt. Mit den Subscribern und den Daten wurde analog verfahren.

Diese Vorgehensweise funktioniert allerdings nur eingeschränkt und weist einige Mängel auf:

- **Connection Handling**
  Die Verbindung zwischen den beiden Endpunkten ist fragil. Es gibt keine zuverlässige Methode um über die uMundo API herauszufinden, ob eine Verbindung mit dem entfernten Endpunkt der Bridge fehlgeschlagen ist und warum. Außerdem muss die Firewall auf beiden Seiten der Bridge so konfiguriert werden, dass eingehende Verbindungen auf zwei bestimmten TCP-Ports erlaubt sind.

- **Vier Connections zwischen den Bridge-Endpunkten**
  Die Art, wie uMundo die Kommunikation zwischen zwei Nodes aufbaut, erfordert, dass jeder Node sich über zwei TCP-Verbindungen mit dem entfernten Node verbindet und anders herum. Das bedeutet insgesamt vier TCP-Verbindungen und vier offene Ports in der Firewall (zwei auf jeder Seite), was wiederum eine schwierigere Einrichtung der Firewall auf beiden Seiten der Bridge zur Folge hat.

- **Unnötiger Netzwerkverkehr**
  Existieren mehrere Subscriptions für einen Channel, so muss diese Information entweder auf einem extra bridgeinternen Channel von einem Endpunkt der Bridge zum anderen Endpunkt übermittelt werden oder es müssen tatsächlich auch mehrere Subscriptions zwischen den beiden Endpunkten erzeugt werden. Für jede zusätzliche Subscription auf dem selben Channel verdoppelt sich daher die übertragene Datenmenge zwischen den Endpunkten der Bridge. Die gleichen Daten werden mehrfach übertragen.
  Will man dieses Problem umgehen, ist ein tieferer Eingriff in den uMundo Core nötig, der sich in der High-Level-API niederschlagen würde, obwohl diese Funktion (pseudo Subscriptions) eigentlich nur von der Bridge benötigt werden würde.

- **RTP Streaming quasi unmöglich**
  uMundo baut für jedes Streaming über RTP eine eigene UDP-Verbindung auf, Multiplexing ist nicht möglich. Das bedeutet, dass für jede RTP-Verbindung im Vorfeld der verwendete Port bekannt sein und vom Publisher und Subscriber explizit gesetzt werden muss, damit Firewalls zwischen den Bridge-Endpunkten entsprechend konfiguriert werden können. Die vorhandene Software, die RTP Publisher/Subscriber einsetzt, muss aus diesem Grund für die Verwendung über die Bridge explizit angepasst werden. Closed-Source Software, die RTP Streaming nutzt und nicht angepasst werden kann, ist daher gar nicht verwendbar.

- **RTP Metadaten müssen über eigenen Channel übertragen werden**
  Da die Verbindungen zwischen den Endpunkten der Bridge für die Weiterleitung von RTP Kommunikation ebenfalls RTP-basiert sind, können die Metadaten der ursprünglichen RTP-Verbindung nicht auf die andere Seite der Bridge gerettet werden. Leitet man die RTP-Metadaten wie *Timestamp*, *Marker*, *Sequence Number* oder auch *Payload Type* über einen eigenen Kommunikationskanal der Bridge weiter, so müssen die individuellen RTP-

Pakete am anderen Ende der Bridge wieder mit den korrespondierenden Metadaten zusammengeführt werden. Dieser Vorgang ist aber aufwändig und verzögert die eigentliche RTP Kommunikation. Ein solches Vorgehen widerspricht der Intention von RTP, welches gerade für die möglichst verzögerungsfreien Echtzeitkommunikation gedacht ist.

- **Kein RTP Multicast möglich**
  Das RTP Streaming von uMundo kann entweder über Unicast oder Multicast erfolgen (siehe Abschnitt 2.4.2). Da die Verwendung von Multicast über Routergrenzen hinweg aber nicht möglich ist, kann die Bridge Multicast nicht korrekt abbilden. Die Lösung besteht auch hier darin, die erforderlichen Metadaten auf anderem Weg zu übermitteln, mit all den damit verbundenen Nachteilen.

Aufgrund der vielen Nachteile wurde diese Implementierungsvariante fallen gelassen.

5.3 Die endgültige Implementierung

Die endgültige Implementierung umgeht alle Probleme des ersten Implementierungsversuchs, indem sie ein eigenes on-the-wire-Protokoll für die Kommunikation zwischen den beiden Endpunkten der Bridge verwendet. Dieses Protokoll nutzt nur eine einzige TCP-Verbindung für die nicht-RTP Publish-/Subscribe-Kommunikation und einen einzelnen UDP-Port für die RTP Kommunikation. Für beide Verbindungen wird der gleiche Port verwendet.

Für die Übermittlung von Metadaten wie Publish- oder Subscribe-Befehlen sowie für die Weiterleitung von nicht-RTP Kommunikation wird die TCP-Verbindung genutzt. Über UDP werden die Daten eines RTP-Publishers übermittelt. Dieses Vorgehen erhält die Semantik der verbindungslosen RTP Kommunikation auch über die Bridge hinweg. Die Kommunikation von nicht-RTP Publish-/Subscribe-Daten und Publish-/Subscribe-Befehlen erfolgt bei uMundo über TCP. Auch hier wird die Semantik von uMundo bei Verwendung der Bridge erhalten.

5.3.1 Das (on-the-wire) Protokoll der Bridge

Das on-the-wire-Protokoll der Bridge arbeitet in zwei Phasen. In der ersten Phase (Connect) wird versucht eine Verbindung mit dem entfernten Endpunkt der Bridge über TCP und UDP aufzubauen und die Protokollversion geprüft. In der zweiten Phase (Relay) wird dann die uMundo Kommunikation weitergeleitet. Schafft es die Bridge nicht, in der ersten Phase eine Kommunikation über UDP herzustellen, so wird eine Warnung ausgegeben und RTP Kommunikation von uMundo nicht weitergeleitet. Für nicht-RTP Kommunikation funktioniert die Bridge aber auch, wenn nur eine TCP-Verbindung aufgebaut werden kann.

Abbildung 5.3.1 zeigt den Ablauf der Connect-Phase. Bridge Endpunkt A übernimmt dabei die Rolle des TCP-Servers und wartet auf Verbindungen auf dem konfigurierten TCP- und UDP-Port. Bridge Endpunkt B übernimmt die Rolle des Clients und verbindet sich mittels TCP und UDP mit dem Server. War die Verbindung erfolgreich, so sendet der Client zunächst den String „clientHello“ auf der TCP- und UDP-Verbindung an den Server. Dieser antwortet mit dem String

„serverHello“, sobald er auf der jeweiligen TCP- oder UDP-Verbindung das „clientHello“ empfangen hat. Nach dem Empfang des „clientHello“ auf der TCP-Verbindung wartet der Server allerdings nur acht Sekunden auf das „clientHello“ auf der UDP-Verbindung. Kommt dieser String dort innerhalb dieser acht Sekunden nicht an, so wird angenommen, dass die Kommunikation über UDP nicht möglich ist und die Weiterleitung von RTP Kommunikation deaktiviert. Ein „serverHello“ wird auf der UDP-Verbindung in diesem Fall nicht gesendet. Der Client wartet nach dem Absenden des „clientHello“ auf beiden Verbindungen ebenfalls acht Sekunden lang auf das „serverHello“ des Servers via TCP und UDP.

Beide Endpunkte brechen die Kommunikation ab, wenn nach dem Aufbau der TCP-Verbindung entweder acht Sekunden lang nichts oder nicht der erwartete String empfangen wird. Der Empfang des „falschen“ Strings auf der UDP-Verbindung führt ebenfalls zu einem Abbruch der Verbindung.

In der Relay-Phase werden auf beiden Verbindungen atomische Nachrichten ausgetauscht. Da TCP-Verbindungen streamorientiert sind und die empfangenen Daten nicht paketweise zurückgeben, werden die Nachrichten hier mit einem **uint32_t** eingeleitet, der die Gesamtlänge der folgenden Nachricht angibt. Bei der UDP-Kommunikation entfällt diese Längenangabe, da jedes UDP-Paket nur genau eine Nachricht enthält. Das on-the-wire Format der Nachrichten selbst wird in den Tabellen 5.3.1 und 5.3.2 genauer beschrieben. **uint32_t** und **uint16_t** Werte werden hierbei als Big-Endian über das Netzwerk übertragen, unabhängig davon, welche Architektur die CPU hat, auf der der Bridge Endpunkt läuft.

Datentyp	Anzahl	Name	Beschreibung
uint16_t	1	version	Versionsnummer der Nachricht. Aktuell ist nur die Protokollversion „1“ definiert.
Key-Value Paar	0..n	none	Beliebig viele der in Tabelle 5.3.2 definierten Key-Value Paare.
uint32_t	1	end	Markiert das Ende einer Nachricht und muss den Wert „0“ haben.

Tabelle 5.3.1.: Nachrichtenformat Version 1

Datentyp	Anzahl	Name	Beschreibung
uint32_t	1	keySize	Länge des Keys des Key-Value Paares.
char[]	1..n	key	Char-Array der Länge *keySize*.
uint32_t	1	valueSize	Länge des Wertes des Key-Value Paares.
char[]	1..n	value	Char-Array der Länge *valueSize*.

Tabelle 5.3.2.: Nachrichtenformat Version 1 - Key-Value Paar Kodierung

Da der Inhalt einer Nachricht auf Key-Value Paaren beruht, können alle Informationen, die über die Bridge laufen, in diesem Format übermittelt werden, ohne dass ein neues on-the-wire Format für jede einzelne Nachricht definiert werden muss. Beim Versenden einer Nachricht über das Netzwerk werden daher zuerst die zu übermittelnden Werte aus Tabelle 5.3.4 zu einem String serialisiert, der dann als Wert eines Key-Value Paares dient. Die resultierende Nachricht wird

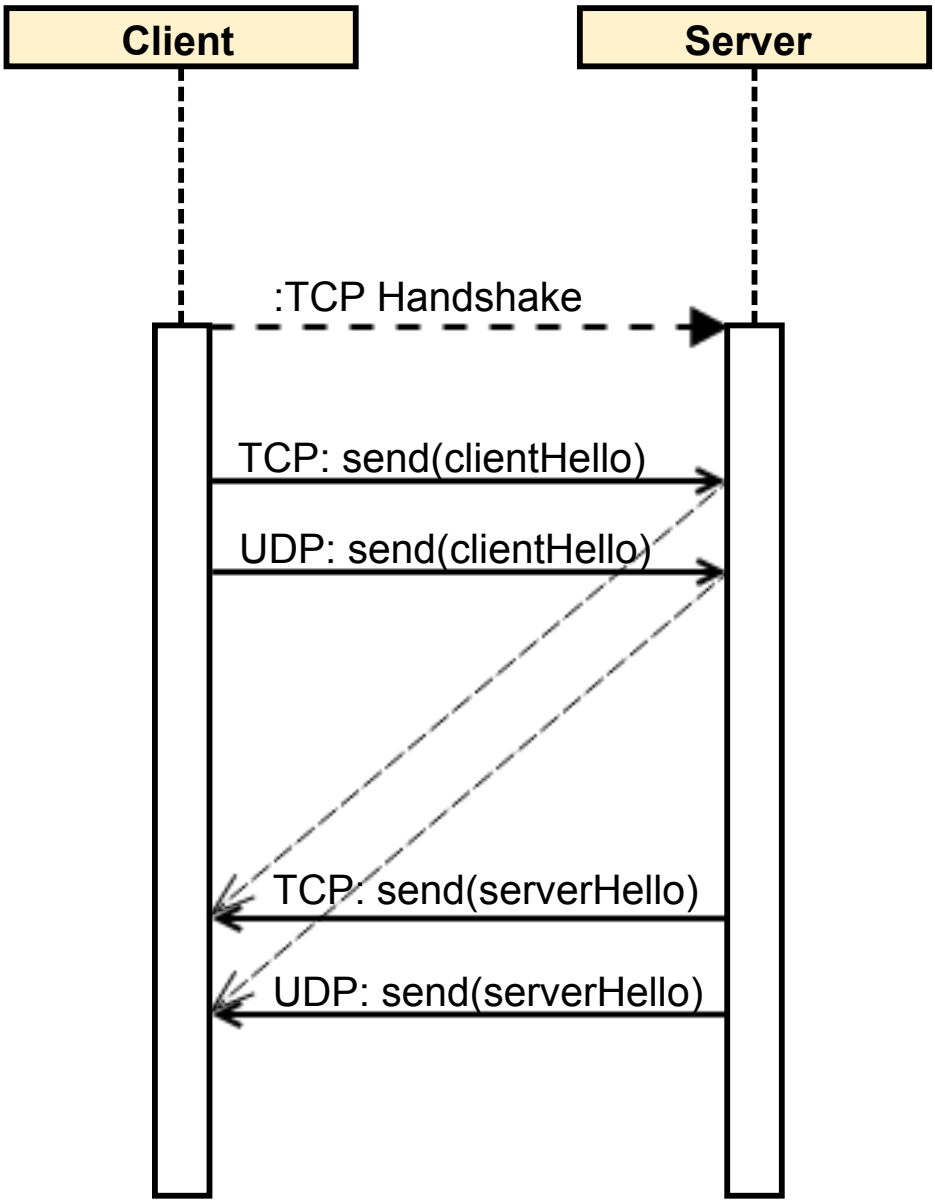


Abbildung 5.3.1.: uMundo Bridge: Ablauf der Connect-Phase

ebenfalls serialisiert (siehe Serialisierungsformat in Tabellen 5.3.1 und 5.3.2) und dann via TCP oder UDP verschickt.

Die Nachrichten mit dem Typ „pubAdded“ und „pubRemoved“ werden versendet, wenn ein neuer Publisher gefunden wird bzw. ein Publisher verschwindet. Die Nachrichten „subAdded“ und „subRemoved“ werden versendet, wenn ein neuer Subscriber gefunden wird bzw. ein Subscriber verschwindet. Die Nachricht „data“ wird gesendet, wenn Daten durch die Bridge weitergeleitet werden. Die Nachricht „internal“ dient der internen Kommunikation zwischen den beiden Bridge Endpunkten. Details zu den einzelnen Nachrichtentypen können in Tabelle 5.3.4 gefunden werden, Tabelle 5.3.3 zeigt die Key-Value Paare (KVP-Felder), die jede Nachricht zwingend enthalten muss.

Key	Beschreibung
type	Gibt den Typ der Nachricht an. Siehe Tabelle 5.3.4 für die Liste der in Protokollversion 1 möglichen Nachrichtentypen und die für diese Typen zusätzlich notwendigen Key-Value Paare.
_source	Wird erst beim Empfang einer Nachricht gesetzt und enthält entweder den Wert „TCPListener“ oder „UDPListener“, je nachdem, ob sie über UDP oder TCP empfangen wurde.

Tabelle 5.3.3.: Nachrichtenformat Version 1 - Pflichtfelder

internal	
Felder	**Beschreibung**
string cause	Bridgeinterne Nachricht. Das Feld „cause“ gibt den Grund der Nachricht an. Der Wert ist entweder „ping“ für einen Ping vom anderen Bridge Endpunkt oder „termination“, falls der TCP oder UDP Empfangsthread beendet wurde.

pubAdded	
Felder	**Beschreibung**
string channelName bool isRTP	Wird gesendet, wenn ein neuer Publisher aufgetaucht ist (siehe Monitor-Interface in Listing 5.1.1, Abschnitt 5.1). „channelName“ gibt dabei den Channel an, auf dem der gefundene Publisher publiziert, „isRTP“ gibt an, ob es sich um einen RTP oder einen normalen Publisher handelt.

pubRemoved	
Felder	**Beschreibung**
string channelName bool isRTP	Wird gesendet, wenn ein Publisher verschwindet. Für die Erklärung der Felder siehe Nachrichtentyp „pubAdded“.

subRemoved	
Felder	**Beschreibung**
string channelName bool isRTP bool isMulticast (string ip) (uint16_t port)	Wird gesendet, wenn ein Subscriber auf dem stellvertretenden Publisher entfernt wird. Für die Erklärung der Felder siehe Nachrichtentyp „subAdded“.

subAdded	
Felder	**Beschreibung**
string channelName bool isRTP bool isMulticast (string ip) (uint16_t port)	Wird gesendet, wenn ein neuer Subscriber auf dem stellvertretenden Publisher auftaucht (siehe Greeter-Interface in Listing 5.1.2, Abschnitt 5.1). „channelName“ gibt dabei den Channel an, um den es sich handelt. „isRTP“ gibt an, ob es sich um einen RTP oder einen normalen Subscriber handelt. Ist „isMulticast“ auf true gesetzt, so müssen die optionalen Felder „ip“ und „port“ gesetzt sein, die angeben, um welche Multicast IP (Multicast Gruppe) es sich handelt und auf welchem Port die Kommunikation stattfinden soll. „isMulticast“ darf nur true sein, wenn auch „isRTP“ auf true gesetzt ist.

data	
Felder	**Beschreibung**
string channelName bool isRTP string data uint32_t metadataCount metadataKey.i metadataValue.i	Wird gesendet, wenn der stellvertretende Subscriber Daten empfängt. „channelName“ gibt dabei den Channel an, um den es sich handelt, „isRTP“ gibt an, ob es sich um RTP oder normale Daten handelt. „data“ enthält die Daten, die vom Publisher gesendet wurden. Die dabei gesetzten Metadaten werden in den Feldern „metadataKey.i“ und „metadataValue.i“ übertragen, wobei das „i“ im Feldnamen bei Null beginnt und für jedes Key-Value Paar der Metadaten um Eins erhöht wird. Die Gesamtzahl der Metadaten wird in „metadataCount“ übertragen. Hinweis: Für RTP Publisher erlaubt uMundo kein Setzen von eigenen Metadaten. Die Bridge überträgt hier allerdings trotzdem die von uMundo automatisch gesetzten Metadaten, welche dem Subscriber auf der anderen Seite der Bridge damit ebenfalls zur Verfügung stehen (wie bei einer direkten Kommunikation auch).

Tabelle 5.3.4.: Nachrichtenformat Version 1 - Nachrichtentypen

5.3.2 Die Komponenten der Bridge

Die Implementierung der Bridge besteht aus mehreren Klassen. Im Klassendiagramm in Abbildung 5.3.2 werden die Abhängigkeiten dieser Klassen untereinander genauer dargestellt.
Die wichtigsten Klassen sind die Klasse „Connector“, die für die Connect-Phase der Bridge zuständig ist und die Klasse „ProtocolHandler“, welche die Relay-Phase der Bridge behandelt. Der *ProtocolHandler* wird durch die Klassen „TCPListener“ und „UDPListener“ unterstützt. Die beiden Listener besitzen jeweils einen eigenen Thread und empfangen die einzelnen Nachrichten über TCP oder UDP. Die empfangenen Nachrichten stellen die beiden Listener dann fertig geparst als Objekt der Klasse „BridgeMessage“ in die Queue „MessageQueue“. Hier kann der *ProtocolHandler* die einzelnen Nachrichten dann von seinem eigenen Thread aus abholen. Die Klasse „BridgeMessage“ dient der Modellierung einer einzelnen Nachricht und kümmert sich sowohl um die Serialisierung der in ihr gespeicherten Key-Value Paare als auch um deren Deserialisierung.
Die Klassen „SocketException“ und „BridgeMessageException“ sind nicht im Klassendiagramm dargestellt. Sie dienen nur der einfacheren Fehlerbehandlung. Die Klassen „GlobalGreeter“ und „PubMonitor“ implementieren das schon erwähnte Greeter-Interface und das neu hinzugefügte Publisher-Monitor-Interface (siehe Abschnitt 5.1). Die Klasse „GlobalReceiver“ stellt die Callback-Klasse für den Datenempfang via uMundo dar (siehe Abschnitt 2.4.1). Diese drei Klassen dienen nur als Verbindungsglied zwischen uMundo und dem *ProtocolHandler*. Zur besseren Lesbarkeit des Codes wurde die Funktionalität dieser Klassen nicht direkt in den *ProtocolHandler* eingebaut.

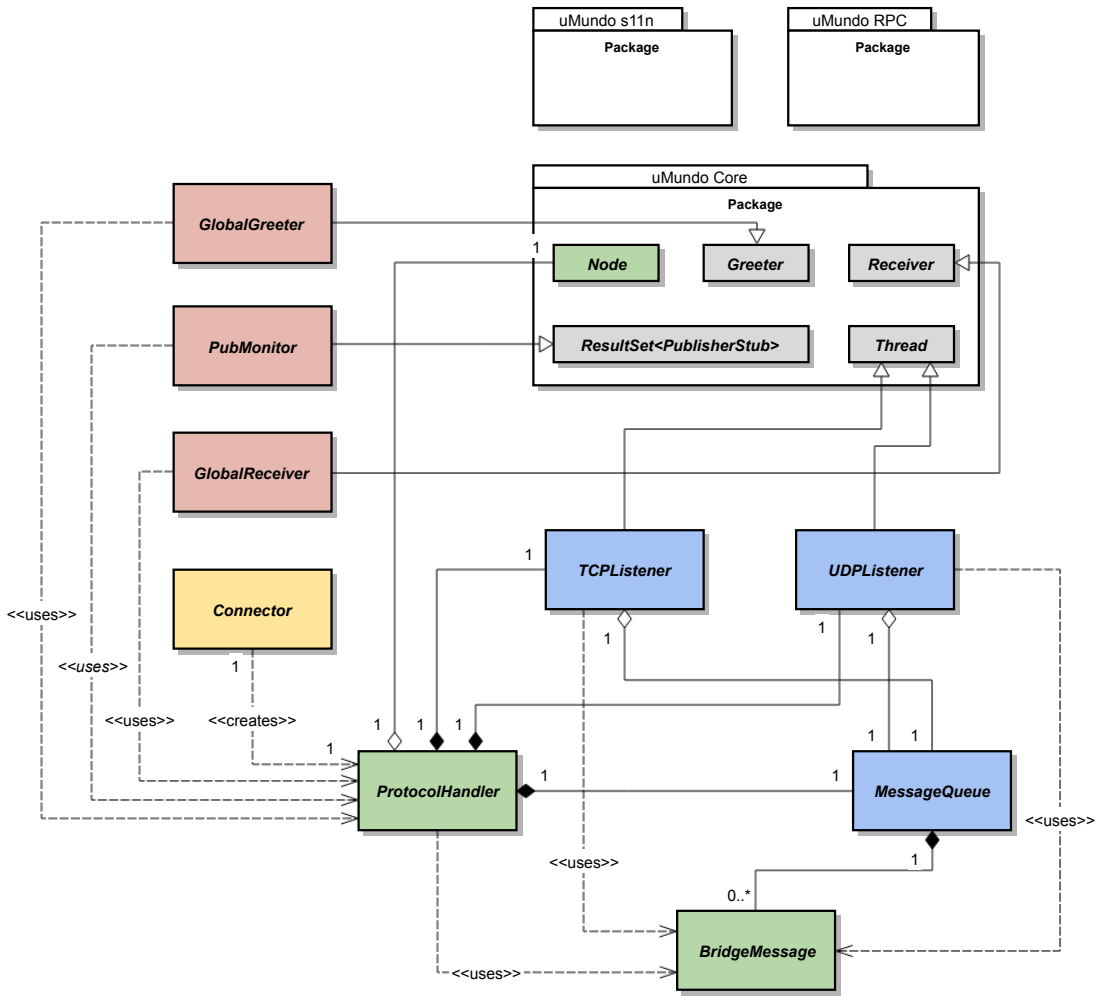


Abbildung 5.3.2.: uMundo Bridge: Klassendiagramm der wichtigsten Klassen

5.4 Hybrid: Bridge + VPN

Wie in Abschnitt 4.3 erklärt, besitzt die Bridge keine Verschlüsselung oder Authentifizierung. Diese Funktionen können aber durch ein Peer-to-Peer VPN nachgerüstet werden. Hierfür wird nach dem Aufbau einer VPN-Verbindung automatisch die Bridge gestartet, die dann durch den VPN-Tunnel kommuniziert. Dadurch ist die Kommunikation der Bridge mittels VPN um Verschlüsselung und Authentifizierung „nachgerüstet“ und kann sicher über das Internet oder andere potentiell unsichere Netzwerke hinweg verwendet werden.

Das Konfigurationsbeispiel eines solchen VPNs findet sich in den Listings A.4.1 und A.4.4.

6 Zusammenfassung

Die Verbindung mehrerer uMundo Knoten über Routergrenzen hinweg ist aufgrund der Verwendung von mDNS und DNS-SD für die Discovery in uMundo nicht möglich. Will man mit uMundo eine Kommunikation zwischen Teilnehmern realisieren, die sich in verschiedenen Netzwerken befinden, so ist das nicht ohne Weiteres möglich.
Ein lösendes Konzept ist die Verwendung eines VPNs, welches die Kommunikation von uMundo und dessen Multicast-Pakete, die für mDNS nötig sind, zwischen den beteiligten Netzwerken hin und her leitet. Ein solches VPN ist allerdings kompliziert einzurichten und weist dabei auch einige Nachteile auf (siehe Abschnitt 4.2).
Da ein VPN nicht ideal ist, wurde in dieser Arbeit ein weiteres Konzept diskutiert: die Verwendung eines Proxys (hier Bridge genannt). Eine solche Bridge lässt sich so implementieren, dass nur die für uMundo nötige Kommunikation weitergeleitet wird. Die Nachteile eines VPNs verschwinden durch die Verwendung einer solchen Bridge vollständig (siehe Abschnitt 4.3).
Möchte man die Bridge noch um Verschlüsselung und Authentifizierung erweitern, so kann man die Kommunikation zwischen den Endpunkten der Bridge durch einen VPN-Tunnel schicken. Die Anforderungen an ein solches VPN sind viel geringer als an ein VPN, welches die uMundo Kommunikation ohne zusätzliche Bridge ermöglicht. Es ist daher viel einfacher einzurichten. Bei diesem Konzept werden die Vorteile einer Bridge mit dem Vorteil eines VPNs (Verschlüsselung und Authentifizierung) kombiniert.

Die im Rahmen dieser Arbeit implementierte Bridge erlaubt im Serverbetrieb eingehende Verbindungen auf jeder IP-Adresse des Rechners, auf dem sie läuft. Besitzt der Rechner mehrere Netzwerkkarten, so werden auch auf all den dazugehörigen IP-Adressen Verbindungen angenommen. Möchte man die Bridge nur an eine bestimmte IP-Adresse binden, so muss der Code der Bridge entsprechend angepasst werden.

Der Overhead der drei in dieser Arbeit diskutierten Konzepte fällt gering aus und liegt bei der Übermittlung von einem Kilobyte Daten bei schätzungsweise 150 Byte (etwa 15%). Die genaue Evaluierung des Overheads bei Verwendung der drei hier vorgestellten Konzepte ist daher Teil einer zukünftigen Arbeit.

Die im Rahmen dieser Arbeit implementierte Bridge erlaubt im Serverbetrieb nur eine einzelne Verbindung von einem entfernten Bridge Endpunkt. Will man mehrere Brücken zu verschiedenen Netzwerken aufbauen, so muss für jedes der entfernten Netzwerke eine eigene Serverinstanz des Programms gestartet werden (siehe auch Abbildung 4.3.1d). Jede dieser Instanzen benötigt einen eigenen TCP-/UDP-Port. Eine denkbare Erweiterung der Bridge wäre es, auf einer Serverinstanz der Bridge mehrere Verbindungen von entfernten Endpunkten zu erlauben.

7 Literaturverzeichnis

[3cl99] University of California, Berkeley: *The BSD 3-Clause License*. http://opensource.org/licenses/BSD-3-Clause. Version: 1999, Abruf: 15.12.2014

[ava] The Avahi Team: *Avahi*. http://www.avahi.org, Abruf: 15.12.14

[Bec11] Becker, Dirk: *OpenVPN - das Praxisbuch ; [Installation, Konfiguration, Administration ; Authentisierung und Verschlüsselung ; Optionen, Tipps und Praxisbeispiele ; inkl. Troubleshooting]*. 2. Aufl. Bonn : Galileo Press, 2011. – ISBN 978–3–836–21671–5

[bon] Apple Inc.: *Bonjour for Developers - Apple Developer*. https://developer.apple.com/bonjour/index.html, Abruf: 15.12.14

[Bra14] Bray, T.: *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159 (Proposed Standard). http://www.ietf.org/rfc/rfc7159.txt. Version: März 2014 (Request for Comments)

[CK13a] Cheshire, S. ; Krochmal, M.: *DNS-Based Service Discovery*. RFC 6763 (Proposed Standard). http://www.ietf.org/rfc/rfc6763.txt. Version: Februar 2013 (Request for Comments)

[CK13b] Cheshire, S. ; Krochmal, M.: *Multicast DNS*. RFC 6762 (Proposed Standard). http://www.ietf.org/rfc/rfc6762.txt. Version: Februar 2013 (Request for Comments)

[dds] Object Management Group: *Data Distribution Service Portal - The Open, Multiplatform, Interoperable Publish-Subscribe Middleware Standard*. http://portals.omg.org/dds/, Abruf: 07.12.2014

[FGM+99] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). http://www.ietf.org/rfc/rfc2616.txt. Version: Juni 1999 (Request for Comments). – Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585

[gpl07] Free Software Foundation: *GNU General Public License, version 3*. http://www.gnu.org/licenses/gpl.html. Version: Juni 2007, Abruf: 04.12.2014

[Hin] Hintjens, Pieter: *ZeroMQ - The Guide*. http://zguide.zeromq.org/page:all, Abruf: 09.12.2014

[JBB92] Jacobson, V. ; Braden, R. ; Borman, D.: *TCP Extensions for High Performance*. RFC 1323 (Proposed Standard). http://www.ietf.org/rfc/rfc1323.txt. Version: Mai 1992 (Request for Comments). – Obsoleted by RFC 7323

[KS05] Kent, S. ; Seo, K.: *Security Architecture for the Internet Protocol*. RFC 4301 (Proposed Standard). http://www.ietf.org/rfc/rfc4301.txt. Version: Dezember 2005 (Request for Comments). – Updated by RFC 6040

[lgp07] Free Software Foundation: *GNU General Public License, version 3*. http://www.gnu.org/licenses/lgpl.html. Version: Juni 2007, Abruf: 07.12.2014

[opea] OpenVPN Technologies, Inc.: *OpenVPN Community Software*. https://openvpn.net/index.php/open-source.html, Abruf: 17.12.14

[opeb] OpenVPN e.V.: *Webseite*. http://www.openvpn.eu, Abruf: 28.11.2014

[osi94] International Telecommunication Union - Telecommunication Standardization Sector of ITU: *Open Systems Interconnection - Model and Notation, ITU-T Recommendation X.200*. http://handle.itu.int/11.1002/1000/2820. Version: Juli 1994, Abruf: 23.12.14

[Pos80] Postel, J.: *User Datagram Protocol*. RFC 768 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc768.txt. Version: August 1980 (Request for Comments)

[Pos81] Postel, J.: *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc793.txt. Version: September 1981 (Request for Comments). – Updated by RFCs 1122, 3168, 6093, 6528

[pro] Google Inc.: *Protocol Buffers - Google Developers*. https://developers.google.com/protocol-buffers/, Abruf: 15.12.14

[PW10] Perkins, C. ; Westerlund, M.: *Multiplexing RTP Data and Control Packets on a Single Port*. RFC 5761 (Proposed Standard). http://www.ietf.org/rfc/rfc5761.txt. Version: April 2010 (Request for Comments)

[SC03] Schulzrinne, H. ; Casner, S.: *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 3551 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc3551.txt. Version: Juli 2003 (Request for Comments). – Updated by RFCs 5761, 7007

[SCFJ03] Schulzrinne, H. ; Casner, S. ; Frederick, R. ; Jacobson, V.: *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc3550.txt. Version: Juli 2003 (Request for Comments). – Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164

[syn] GitHub User psy0rz: *Advanced event framework in C++*. http://open.syn3.nl/syn3/trac/default/wiki/projects/synapse, Abruf: 04.12.2014

[Ter13] TERRIBERRY, T.: *Update to Remove DVI4 from the Recommended Codecs for the RTP Profile for Audio and Video Conferences with Minimal Control (RTP/AVP)*. RFC 7007 (Proposed Standard). `http://www.ietf.org/rfc/rfc7007.txt`. Version: August 2013 (Request for Comments)

[wik] OpenVPN e.V.: *Wiki*. `http://wiki.openvpn.eu`, Abruf: 28.11.2014

[Zel08] ZELLER, Thomas: *OpenVPN kompakt* -. Saarbr : Brain-Media.de, 2008. – ISBN 978–3–939–31651–0

8 Abbildungsverzeichnis

9 Tabellenverzeichnis

10 Listing-Verzeichnis

Anhang

A Listings

In diesem Kapitel finden sich vollständige Minimalbeispiele für viele in dieser Arbeit erwähnte Konfigurationen oder Codebeispiele.

A.1 umundo.core: Einfaches Publish/Subscribe via Messages

Der Publisher:

```
1  #include "umundo/core.h"
2  #include <iostream>
3  #include <string.h>
4
5  using namespace umundo;
6
7  int main(int argc, char** argv) {
8      //create discovery (mdns) and add a new communication node to it
9      Discovery disc(Discovery::MDNS);
10     Node node;
11     disc.add(node);
12
13     //create publisher for channel "mychannel" and add it to the node
14     Publisher pubFoo("mychannel");
15     node.addPublisher(pubFoo);
16
17     //send message containing "pingexample" every 1000 milliseconds
18     while(1) {
19         Thread::sleepMs(1000);
20         Message* msg = new Message();
21         msg->setData("pingexample", 11);
22         std::cout << "o" << std::flush;
23         pubFoo.send(msg);
24         delete(msg);
25     }
26 }
```

Listing A.1.1: Full Listing: uMundo API für simple Publish/Subscribe Anwendungen - Publisher

Der Subscriber:

```
1  #include "umundo/core.h"
2  #include <iostream>
3
4  using namespace umundo;
5
6  //simple callback class which receives incoming messages
7  class TestReceiver : public Receiver {
8  public:
9      TestReceiver() {};
10     void receive(Message* msg) {
11         std::cout << "i(" << msg->data() << ")" << std::endl << std::flush;
12     }
13 };
14
15 int main(int argc, char** argv) {
16     //create discovery (mdns) and add a new communication node to it
17     Discovery disc(Discovery::MDNS);
18     Node node;
19     disc.add(node);
20
21     //create a new callback object (receiver)
22     TestReceiver* testRecv = new TestReceiver();
23
24     //create subscriber for channel "mychannel" and add it to the node
25     Subscriber subFoo("mychannel", testRecv);   //use testRecv callback ↩
           object
26     node.addSubscriber(subFoo);
27
28     //the receiver callback will be called from another thread
29     //we don't have to do anything more in this thread
30     while(1)
31         Thread::sleepMs(4000);
32 }
```

Listing A.1.2: Full Listing: uMundo API für simple Publish/Subscribe Anwendungen - Subscriber

A.2 umundo.core: Einfaches Streaming via RTP

Der Publisher:

```
1  #include "umundo/core.h"
2  #include <iostream>
3  #include <string.h>
4
5  using namespace umundo;
6
7  int main(int argc, char** argv) {
8      //create discovery (mdns) and add a new communication node to it
9      Discovery disc(Discovery::MDNS);
10     Node node;
11     disc.add(node);
12
13     //create publisher for channel "mysoundchannel" and add it to the node
14     RTPPublisherConfig pubConfig(1, 96);        //user defined data (payload↩
           type 96, TimestampIncrement 1)
15     Publisher pubFoo(Publisher::RTP, "mysoundchannel", &pubConfig);
16     node.addPublisher(pubFoo);
17
18     //send messages cycling through the alphabet (instead of real sound)
19     uint16_t num=0;
20     char buf[2]="A";
21     while(1) {
22         Thread::sleepMs(1000);
23         //cycle through the alphabet (capital letters)
24         buf[0]=65+num++;
25         if(num==26)
26             num=0;
27         Message* msg = new Message();
28         std::string ping=std::string("ping-")+std::string(buf);
29         msg->setData(ping.c_str(), ping.length());
30         std::cout << "o-" << buf << std::endl << std::flush;
31         pubFoo.send(msg);
32         delete(msg);
33     }
34
35     return 0;
36 }
```

Listing A.2.1: Full Listing: uMundo API für simple RTP Publish/Subscribe Anwendungen - Publisher

Der Unicast Subscriber:

```
1  #include "umundo/core.h"
2  #include <iostream>
3  #include <string.h>
4
5  using namespace umundo;
6
7  //simple callback class which receives incoming messages
8  class TestReceiver : public Receiver {
9  public:
10     TestReceiver() {};
11     void receive(Message* msg) {
12         std::string data(msg->data(), msg->size());
13         std::cout << "RTP(" << msg->size() << ") -> '" << data << "'" << ↩
               std::endl << std::flush;
14     }
15 };
16
17 int main(int argc, char** argv) {
18     //create discovery (mdns) and add a new communication node to it
19     Discovery disc(Discovery::MDNS);
20     Node node;
21     disc.add(node);
22
23     //create a new callback object (receiver)
24     TestReceiver testRecv;
25
26     //create unicast subscriber for channel "mysoundchannel" and add it to ↩
           the node
27     RTPSubscriberConfig subConfig;
28     Subscriber subFoo(Subscriber::RTP, "mysoundchannel", &testRecv, &↩
           subConfig);
29     node.addSubscriber(subFoo);
30
31     while(1)
32         Thread::sleepMs(4000);
33     return 0;
34 }
```

Listing A.2.2: Full Listing: uMundo API für simple RTP Publish/Subscribe Anwendungen - Unicast Subscriber

Der Multicast Subscriber:

```
1  #include "umundo/core.h"
2  #include <iostream>
3  #include <string.h>
4
5  using namespace umundo;
6
7  //simple callback class which receives incoming messages
8  class TestReceiver : public Receiver {
9  public:
10     TestReceiver() {};
11     void receive(Message* msg) {
12         std::string data(msg->data(), msg->size());
13         std::cout << "RTP(" << msg->size() << ") ---> '" << data << "'" << ↩
               std::endl << std::flush;
14     }
15 };
16
17 int main(int argc, char** argv) {
18     //create discovery (mdns) and add a new communication node to it
19     Discovery disc(Discovery::MDNS);
20     Node node;
21     disc.add(node);
22
23     //create a new callback object (receiver)
24     TestReceiver testRecv;
25
26     //create multicast subscriber for channel "mysoundchannel" and add it to↩
           the node
27     RTPSubscriberConfig subConfig;
28     subConfig.setMulticastPortbase(42042);
29     subConfig.setMulticastIP("239.1.2.3");      //not needed (default ↩
           multicast group: 239.8.4.8)
30     Subscriber subFoo(Subscriber::RTP, "mysoundchannel", &testRecv, &↩
           subConfig);
31     node.addSubscriber(subFoo);
32
33     while(1)
34         Thread::sleepMs(4000);
35     return 0;
36 }
```

Listing A.2.3: Full Listing: uMundo API für simple RTP Publish/Subscribe Anwendungen - Multicast Subscriber

A.3 VPN

Die Datei „secret.key“ enthält den auf Client und Server verwendeten Verschlüsselungsschlüssel und kann mit **openvpn ––genkey ––secret secret.key** erzeugt werden. Der Schlüssel muss auf Client und Server identisch sein.

```
1  #listen on udp port 1194 for client
2  mode p2p
3  proto udp
4  port 1194
5  float   #allow peer to change ip addresses while communicating
6
7  #first IP is local, second IP is remote
8  ifconfig 10.4.0.1 10.4.0.2
9
10 #maximum transfer unit
11 tun-mtu 1500
12 mssfix
13
14 #virtual network device type (tap tunnels ethernet frames, tun tunnels only ↩
       ip packets)
15 dev tun
16
17 #encription configuration
18 secret secret.key
19 cipher AES-256-CBC
20
21 #compress everything for transmission
22 comp-lzo
23
24 #loglevel
25 verb 3
26
27 #ping every 5 seconds, restart VPN tunnel, if no ping succeeded for 16 ↩
       seconds
28 ping 5
29 ping-restart 16
30 ping-timer-rem  #ping only if we are connected
```

Listing A.3.1: Full Listing: Peer to Peer VPN Server

```
1  #connect to vpn-server.example.com on port 1194 via udp
2  mode p2p
3  proto udp
4  remote vpn-server.example.com 1194
5  nobind  #don't bind to local address and port
6  float   #allow peer to change ip addresses while communicating
7
8  #first IP is local, second IP is remote
9  ifconfig 10.4.0.2 10.4.0.1
10
11 #maximum transfer unit
12 tun-mtu 1500
13 mssfix
14
15 #virtual network device type (tap tunnels ethernet frames, tun tunnels only ↩
       ip packets)
16 dev tun
17
18 #encription configuration
19 secret secret.key
20 cipher AES-256-CBC
21
22 #compress everything for transmission
23 comp-lzo
24
25 #loglevel
26 verb 3
27
28 #ping every 5 seconds, restart VPN tunnel, if no ping succeeded for 16 ↩
       seconds
29 ping 5
30 ping-restart 16
31 ping-timer-rem  #ping only if we are connected
```

Listing A.3.2: Full Listing: Peer to Peer VPN Client

A.4 Hybrid: Bridge + VPN

Die Datei „secret.key“ enthält den auf Client und Server verwendeten Verschlüsselungsschlüssel und kann mit **openvpn ––genkey ––secret secret.key** erzeugt werden. Der Schlüssel muss auf Client und Server identisch sein.

```
1  #listen on udp port 1194 for client
2  mode p2p
3  proto udp
4  port 1194
5  float   #allow peer to change ip addresses while communicating
6
7  #first IP is local, second IP is remote
8  ifconfig 10.4.0.1 10.4.0.2
9
10 #maximum transfer unit
11 tun-mtu 1500
12 mssfix
13
14 #virtual network device type (tap tunnels ethernet frames, tun tunnels only ↩
       ip packets)
15 dev tun
16
17 #encription configuration
18 secret secret.key
19 cipher AES-256-CBC
20
21 #compress everything for transmission
22 comp-lzo
23
24 #loglevel
25 verb 3
26
27 #ping every 5 seconds, restart VPN tunnel, if no ping succeeded for 16 ↩
       seconds
28 ping 5
29 ping-restart 16
30 ping-timer-rem  #ping only if we are connected
31
32 #automatically run  bridge server on client connect
33 up server_up.sh
34 up-delay
35 down server_down.sh
36 down-pre
37 script-security 2   #allow script execution
```

Listing A.4.1: Full Listing: Hybrid Bridge + VPN Server Config

```
1 #!/bin/sh
2
3 ../../umundo/build/bin/umundo-bridge -v -l 5555 &
4 echo $! > bridge_server.pid
5 exit 0
```

Listing A.4.2: Full Listing: Hybrid Bridge + VPN Server (Zusatzskript server_up.sh)

```
1 #!/bin/sh
2
3 kill $(cat bridge_server.pid)
4 rm bridge_server.pid
5 exit 0
```

Listing A.4.3: Full Listing: Hybrid Bridge + VPN Server (Zusatzskript server_down.sh)

```
#connect to vpn-server.example.com on port 1194 via udp
mode p2p
proto udp
remote localhost 1194
nobind  #don't bind to local address and port
float   #allow peer to change ip addresses while communicating

#first IP is local, second IP is remote
ifconfig 10.4.0.2 10.4.0.1

#maximum transfer unit
tun-mtu 1500
mssfix

#virtual network device type (tap tunnels ethernet frames, tun tunnels only ip packets)
dev tun

#encription configuration
secret secret.key
cipher AES-256-CBC

#compress everything for transmission
comp-lzo

#loglevel
verb 3

#ping every 5 seconds, restart VPN tunnel, if no ping succeeded for 16 seconds
ping 5
ping-restart 16
ping-timer-rem  #ping only if we are connected

#automatically run  bridge client after successful vpn connect
up client_up.sh
up-delay
down client_down.sh
down-pre
script-security 2   #allow script execution
```

Listing A.4.4: Full Listing: Hybrid Bridge + VPN Client Config

```
1 #!/bin/sh
2
3 ../../umundo/build/bin/umundo-bridge -v -c ${ifconfig_remote}:5555 &
4 echo $! > bridge_client.pid
5 exit 0
```

Listing A.4.5: Full Listing: Hybrid Bridge + VPN Client (Zusatzskript client_up.sh)

```
1 #!/bin/sh
2
3 kill $(cat bridge_client.pid)
4 rm bridge_client.pid
5 exit 0
```

Listing A.4.6: Full Listing: Hybrid Bridge + VPN Client (Zusatzskript client_down.sh)